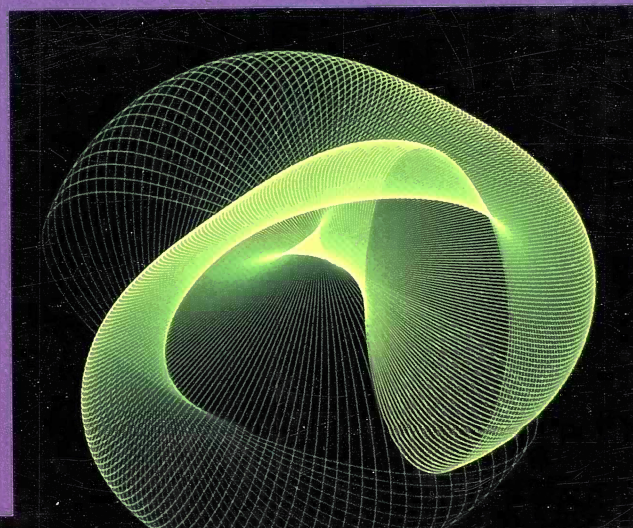


# OS/2 2.1<sup>TM</sup>

# WORKPLACE SHELL PROGRAMMING

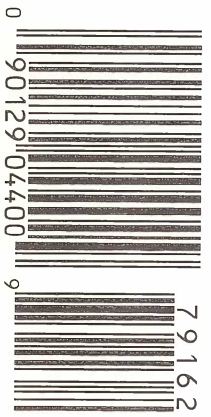
Stefano Maruzzi



RANDOM HOUSE  
ELECTRONIC PUBLISHING

3½" Disk  
Included






# **OS/2 2.1 Workplace Shell Programming**



---

# OS/2 2.1 Workplace Shell Programming



*Stefano Maruzzi*



OS/2 2.1 Workplace Shell Programming

Copyright © 1994 by Stefano Maruzzi

Composed and produced by SunCliff Graphic Productions

All rights reserved. No part of the contents of this book may be reproduced in any form or by any means without the written permission of the publisher.

Published in the United States by Random House, Inc., New York, and simultaneously in Canada by Random House of Canada, Limited.

Manufactured in the United States of America.

First Edition

0 9 8 7 6 5 4 3 2 1

ISBN 0-679-79162-0

The author and publisher have used their best efforts in preparing this book, the disk accompanying this book, and the programs and data contained herein. However, the author and publisher make no warranties of any kind, express or implied, with regard to the documentation or programs or data contained in this book or disk, and specifically disclaim, without limitation, any implied warranties of merchantability and fitness for a particular purpose with respect to the disk, the programs, and/or the techniques described in the book. In no event shall the author or publisher be responsible or liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or any other damages in connection with or arising out of furnishing, performance, or use of this book or the programs or data.

#### **Trademarks**

A number of entered words in which we have reason to believe trademark, service mark, or other proprietary rights may exist have been designated as such by use of initial capitalization. However, no attempt has been made to designate as trademarks or service marks all personal computer words or terms in which proprietary rights might exist. The inclusion, exclusion, or definition of a word or term is not intended to affect, or express any judgment on, the validity or legal status of any proprietary right, which may be claimed in that word or term.

New York Toronto London Sydney Auckland

# Preface

Today, PC users demand not just simple GUI applications, but applications that must also be powerful, easy to use, fast, reliable, and, above all, rich in features. This book shows how to write this kind of program using the OS/2 2.1 Presentation Manager API and Workplace Shell (WPS) objects. Starting from the ground up, you will learn the development model in great detail, adding new concepts to the general scheme. At the end of the reading, you will be able to write the kinds of professional applications that the current market demands.

This book is the result of more than six years of working in OS/2 since the release of version 1.0. It has been written from scratch based on the 2.1 API, closely following the user interface rules introduced by Workplace Shell and the object-oriented interface. It not only explains how to write OS/2 applications, but also how to create a user-friendly interface exploiting all the WPS features, such as drag & drop, direct manipulation of objects, and extensive use of the pointer device. Each subject has its own collection of examples with which to put theory into practice. The source code samples (over 1.5 MB of data) clarify the use of the API and are a good basis for writing customized software. In the menu chapter, for example, you will learn how to create a basic menu bar and modify its attributes. You will also see how to create run-time menus, window context menus, and run-time pop-up menus. Drag & drop and WPS objects are the topics of the last two chapters. Detailed examples illustrate how to implement titlebar dragging and write applications that interact with common objects, such as Font Palette and Color Palette.

The next generation of operating systems will make use of an object-oriented shell, and OS/2 2.x has been among the first widely used operating systems to bring this capability to the market. The direct manipulation of objects requires the development of a new breed of applications deeply integrated in the system shell. As a matter of fact, in the near future the distinction between the operating system and third-party applications will begin to fade, eventually moving to a fully object-oriented operating environment. Therefore, the implementation of drag & drop is a real must to extend the system shell behavior into every software program. The next and final step coincides with the use of objects provided by the system.

The WPS compliant editor described in Chapter 13 is good example of the future direction of software development under OS/2 2.x. The integration of hand-written

code and system object is seamless, reducing the burden of procedural coding. In PMEDIT you can drag objects from e to the shell, easing typical applications such as opening and saving a document.

To further enhance your use of this book, we have placed all the listings on the accompanying disk, both to save trees and because this is how (where?) you'll really be using the listings. Detailed instructions on how to use the disk are provided on the last two pages of the book.



*This book is dedicated to my wife Antonella*

## *Acknowledgments*

I have always dreamt of writing a technical book in English, and thanks to Steve Guty and the support of Jean Davis-Taft and Michael Aquilante of Random House, this dream has become a reality. I hope this is just the beginning of a long and successful working relationship, with new titles to come.

I would also like to thank Stefano Tendon, Claudio Galletti, and Marco Cantù for their work, support, suggestions, and help.

Stefano Maruzzi can be reached on these electronic e-mail systems:

MCI Mail: Stefano Maruzzi (555-4412)

CompuServe: 100115,356



# Contents

Preface v

Acknowledgments vii

---

## **1 *An Introduction to Presentation Manager*** 1

- OS/2 Applications 1
- Multitasking in OS/2 5
- The PM Screen Group 8
  - Working in the PM Screen Group 8
- The API of OS/2 8
- The Development of PM Applications 11
  - Software Tools 11
- The C Language and PM 12
  - Header Files 12
  - Handles 13
  - Data Types and the Defines in PM 15
  - The EXPENTRY Functions 17
- Debugging in OS/2 PM 19
- Kwikinf 20

---

## **2 *The Development Model*** 21

- CUA 89 and the System Menu 23
- CUA 91 and the System Menu 23
- The Sizing Icons 24
- The Menu Bar 24
- Flow Chart 24
- The Make File 26
- Resource Files 31

- Header Files 34
- The Source Code 35
- The *Main()* Function 36
  - Initialization of an Instance 37
  - Creating the Message Queue 39
  - Registering a Window Class 39
  - Class Styles 41
  - Window Words 41
- The Predefined Window Classes 43
- Why Register Window Classes? 45
  - The Nature of PM 45
  - Strategies for Registering Window Classes 46
- Creating a Window 46
  - The Parameters of *WinCreateStdWindow()* 47
  - Comparison between FS\_ and FCF\_ Styles 53
- Some Remarks on *WinCreateStdWindow()* 56
  - Errors with *WinCreateStdWindow()* 56
- Displaying a Window 56
- The Message Loop 60
  - Living in the Message Loop 64
  - Execution Termination 64
- The *Main()* Function for a Generic PM Application 65
- The Window Procedure 66
  - A Sample Application 67
  - Some Alternate Solutions and Enhancements 71
  - The Look of Windows 72
  - The Application Title 72

---

### 3 *Messages* 75

- Painting 78
  - Output Techniques in PM 81
  - Distinction between Presentation Space and Device Context 83
- The Presentation Spaces 83
  - The Cached Micro PS 84
  - The Micro PS 85
  - The Normal PS 86

The WM_PAINT Message	87
The Update Region	87
Forcing a WM_PAINT Message	89
Output Synchronization	91
Output Handling	92
Erasing a Window's Background	94
Message Flow in PM	97
Queued Messages	97
The Parameters of <i>WinPostMsg()</i>	100
When to Post a Message	100
Non-Queued Messages	101
When to Send a Message	103
Some Message-Sending Considerations	103
Functions and Messages	104
Messages, Windows, and Window Procedures	107
Functions That Use Messages	108
The Messages of the Predefined Classes	108
Defining New Messages	108
Some Enhancements	109
Message Parameters	109
Sending Messages	111
Execution of Painting	115

---

## 4 *Windowing* 119

Creating a Window with <i>WinCreateStdWindow()</i>	119
The Frame Window	119
Parenthood	120
Sibling Windows	126
Ownership	126
The Frame Control Window	130
Reserved Memory	132
Extending the Reserved Memory Area	135
Querying the Reserved Memory Area	136
Window Words' Usage Rules	137
Message Passing	137
The <i>WinCreateWindow()</i> Function	138
Building Standard Windows with <i>WinCreateWindow()</i>	140

The FCF_ Flags and the WC_FRAME Class	143
Experiments Using <i>WinCreateWindow()</i>	144
The CREATESTRUCT Structure	144
How to Destroy a Window	145
Closing a Window in a Multi-Window Application	146
Sizing and Positioning a Window	147
Saving the Position of a Window	153
Creating a Client's Child Window	154
Informing the Window List	156
Execution of One Single Instance	159
Accessing the Window List	160
Enumerating Top-Level Windows	163
A Third Solution	165
Other Solutions	168
Registering a Public Class	168
The <i>WinMessageBox()</i> Function	169
The Focus Chain	175
Adding a System Icon	179

---

## 5 *Input Tools and Resources* 183

The Keyboard	183
Keyboard Usage	187
The Mouse	188
Teaching an Old Mouse New Mouse Tricks	193
Mouse Clicking	195
Selection of an Object	195
Selecting More Objects	196
Drag & Drop	197
Timers	198
The Resource File	200
The Nature of Resources	201
The Text Resources	202
STRINGTABLE	204
Loading a String	206
Defining Computed IDs	207
Why You Should Use a STRINGTABLE	208

MESSAGETABLE	209
WINDOWTEMPLATE	211
Using Resources	215
Binary Resources	217
Loading an Icon	218
Predefined Icons and Pointers	221
Displaying Predefined Bitmaps	226
Moving an Icon in a Window	228
Moving a Bitmap in a Window	233
ASSOCTABLE	236

---

## 6 *Menus* 241

The Titlebar Menu	243
The Titlebar Menu and WPS	246
The APIs of PM and the System Menu	248
Giving a Menu to a Window	249
Creating a Menu Template	249
The Menu Template	251
Keyboard Access	253
Menu Style Rules	256
Defining Templates	258
Complex Menu Templates	261
Syntax Rules for Menu Templates	264
Loading a Menu Template	265
Menus, Parenthood, and Ownership	268
Modifying the Window Procedure	269
Recognizing the Source of a Selection	275
Changing Attributes Dynamically	278
Messages and Macros	281
Loading a New Menu	281
Bitmaps As Menu Items	285
Menus Built by the Application	286
Accelerators	287
Run-Time Menus	292
WPS Menus	294
Interactions between the Menu Bar and the Client Window	296

Window Context Menu 297  
Detecting WM\_CONTEXTMENU 298  
Emphasizing an Object 303

---

## 7 *The Predefined Window Classes* 307

The Message Flow 309  
Peeping through Windows 310  
The Structure of Windows 316  
The Predefined Window Classes and the Window Words 317  
Undocumented Classes 318  
How to Create a Window of Class WC\_ 318  
    When to Create a Window of a Predefined Class 320  
The Class WC\_BUTTON 320  
    Fashionable Buttons 323  
    The BTNCDATA Structure 324  
    Interacting with a Button 326  
    Notification Codes 327  
    Pushbuttons as Input Elements 327  
The Class WC\_STATIC 329  
    The SS\_Styles 330  
    The SM\_Messages 332  
The Class WC\_TITLEBAR 332  
The Class WC\_SCROLLBAR 333  
    The SBS\_Styles 333  
    The SBM\_Messages 333  
    Some Considerations 337  
The Class WC\_LISTBOX 338  
    The Styles of the Class WC\_LISTBOX 339  
    The LM\_Messages 339  
    The Reserved Memory Area 342  
    Multiple Selection Listboxes 343  
    The Notification Codes 345  
    A Simple Listbox 346  
    Message Flow 349  
    Creating a Listbox 350  
    Owner-Drawn Listboxes 354



- The WM\_MEASUREITEM Message 354
- The WM\_DRAWITEM Message 354
- Creating a Listbox 358
- Handling Information 360
- Drawing an Item 361
- Some Considerations 365
- The Class WC\_ENTRYFIELD 365
  - The ES\_Styles 365
  - The EM\_Messages 366
  - Notification Codes 367
  - The ENTRYFDATA Structure 368
  - The Class WC\_COMBOBOX 369
  - The CBS\_Styles 369
  - The CBM\_Messages 370
  - The Notification Codes 370
  - Using a Combobox 370
- The Class WC\_MLE 372
  - The Styles of the Class WC\_MLE 372
  - The MLM\_Messages 373
  - The MLN\_Notification Codes 377
  - Data Structures of the Class WC\_MLE 377
  - The MLECTLDATA Structure 379
- The Class WC\_NOTEBOOK 380
  - Filling a Notebook 383
  - Inserting a Page 385
  - Associating Information with a Page 387
  - Notification Codes 387
  - A First Try 387
- The Class WC\_CONTAINER 388
  - The Styles of the Class WC\_CONTAINER 388
  - The Logic of the WC\_CONTAINER Class 389
  - Creating a Container 392
  - The Objects of a Container 393
  - Diversions with Containers 403
  - The Window Context Menu 404
  - Proliferation of Objects 406
  - Other Actions on Containers 407
- The Class WC\_SLIDER 410
  - Creating a Slider 414

- Messages of the Class WC\_SLIDER 414
  - A Sample Slider 417
- The Class WC\_SPINBUTTON 417
  - Master or Servant? 421
  - The Notification Codes 422
  - A Sample Spinbutton 422
- The Class WC\_VALUESET 424
  - Creating a Valueset 424

---

## 8 *Dialog Windows* 429

- Two Types of Dialogs 430
  - Features of a Dialog 430
  - Creating a Dialog 432
  - Modal or Modeless? 435
  - Dialog Templates 436
  - The Styles WS\_GROUP and WS\_TABSTOP 438
  - The Dialog Procedure 439
  - A Doubtful Question 439
- The Message WM\_INITDLG 441
  - The Ownership Problem 443
  - Getting to Know the Owner 445
  - Accessing Controls 446
- The Presentation Parameters 448
  - Setting the Presentation Parameters 452
  - Presentation Parameters and *WinCreateWindow()* 454
  - Presentation Parameters and Resource Files 456
  - Terminating a Modal Dialog 456
  - Default Message Processing 457
  - Some Considerations 458
- A Utility for PM 458
  - Searching for a File 459
  - The Scheme of the Application 461
  - The Accelerator Table 463
  - Error Handling 463
  - Executing a File 463
  - Selection of a File 464

- Deleting Files 465
- Searching for Files 468
- Creating an Open Box 469
  - Positioning the Dialog 470
  - A New Data Type 470
  - Centering the Dialog 471
  - Filling in Controls 472
  - Input Sources 472
  - Selecting a File 475
- Predefined Dialogs 476
  - Accessing the File Systems 476
  - Listing Fonts 478
- Modeless Dialogs 480
  - A Sample of a Modeless Dialog 480

## **9** *Developing Fast Multithreaded Applications* 483

- Memory Allocation 485
  - Memory Management 486
- Segmented Applications 487
- Code Segment and Data Segment 487
  - Executing Several Instances 488
  - Producing a Fast Executable 489
  - The Structure of the DEF 490
  - Segmentation Rules 491
- A Menu Editor 492
  - The Interface of Menu Maker 493
  - The Maximized Window 493
  - The Empty Menu Bar 493
  - The Control Panel 495
  - The Application's Logic 495
  - Defining a Top-Level Menu 496
  - Defining a Drop-Down Menu 497
  - Defining a Separator 499
  - Defining a Submenu 500
  - Saving the Template 501
  - Loading a Menu Template 502

- Parsing the MNU and HHH Files 504
- The MENUITEM Structure 505
- The Position 505
- The Style 505
- The Attributes 506
- The ID 506
- The Submenu Handle 506
- The Bitmap Handle 506
- Saving a Menu Template 507
- Adding a Top-Level 507
- Adding a Drop-Down 508
- Adding a SEPARATOR 509
- Adding a Submenu 509
- Gathering Menu Maker Functions 510
- Multithreaded OS/2 Applications 511
  - Creating a Multithreaded Application 511
  - Creating a Thread 512
  - Compiling a Multithreaded Application 514
  - Threads and PM Applications 514
  - The Priority Classes 515
  - Selecting a Priority Class 518
- Multithreaded WHEREIS 519
  - Some Considerations 522
  - Performance Tuning 524
  - Yet More Enhancements 525

---

## **10 *Subclassing, Superclassing, and DLL* 529**

- Accessing the Window Procedure 531
- Performing Subclassing 534
  - A Sample Subclassing 536
  - When to Perform Subclassing 536
- Superclassing 537
  - Performing Superclassing 537
  - Features of Superclassing 539
- Dynamic Linking Libraries 540
  - Definition of a DLL 540
  - How to Produce a DLL 541

- Advantages of DLLs 542
  - Producing a DLL 544
  - Static Linking 544
  - Dynamic Linking 545
  - Structure of a Relocation Record 546
  - Executing a Program That Accesses a DLL 548
  - Loading a DLL Implicitly 548
  - Loading a DLL Explicitly 549
  - Coding a DLL 551
  - Creating a Sample DLL 552
    - Compiling a DLL 554
    - The DEF File of a DLL 554
    - Producing an Import Library 558
  - Creating a New Control 559
  - Creating a New Predefined Class 560
    - Constructing a New Window Class 562
    - Writing the Window Procedure 563
    - Creating a Window of the New Class 564
    - Passing Information 565
    - How to Install a New Class of Controls 567
    - Some Considerations 570
- 

## **11 *Data Sharing Techniques: Clipboard and DDE* 571**

- The Clipboard 571
  - Clipboard Management 572
  - Inserting Data into the Clipboard 574
  - Transferring an Object 576
  - Retrieving the Clipboard's Contents 577
  - Examining the Contents of the Clipboard 578
  - CLIPPUT and CLIPSHOW 578
- Dynamic Data Exchange 582
  - A DDE Conversation 584
  - Designing a DDE Conversation 584
  - Features of a Transaction 585
  - Initiating a DDE Conversation 586
  - Requesting Data 590
  - Providing Data to the Client 593

- Establishing a Permanent Link 595
- Terminating a DDE Conversation 595
- Invisible Windows 596
- Uses of DDE 598
- Defining the Project 598
- The Server 601
- The Client 604
- The Client Interface 604
- Some Considerations 606

---

## **12 *Drag & Drop*** 607

- The Drag & Drop API 607
  - The Logic of Drag & Drop 610
  - Selecting Objects 613
  - Preparing the Image 622
  - Executing the Drag 623
- Preparing Objects for Dropping 625
  - Changing the Cursor's Look 631
  - Acceptance Feedback 633
  - Return Value of DM\_DRAGOVER 634
  - Frame/Client and Dragging 636
  - Receiving Objects 637
  - Dragging the Titlebar Icon 638
- DRAG: Preparing Data 639
- DROP: Accepting an Object 644
  - Intercepting DM\_DROP 646
- Drag & Drop and Valuesets 648
- Drag & Drop and Listboxes 650
- Drag & Drop and Folders 654

---

## **13 *WPS Programming*** 659

- Integrating in WPS 660
  - Minimal Integration 660
  - Medium Integration 660
  - Advanced Integration 660

- How to Develop for OS/2 661
  - Analysis of the Program 668
  - Creating a Panel 674
  - The API for WPS Objects 677
  - A Simple Installation Program 684
  - Destroying an Object 691
- Developing a WPS Editor 691
  - Loading a File 692
  - Editing a New Document from Scratch 692
- Saving a Document 693
  - Printing 694
  - Interacting with the Clipboard 694
  - Searching and Replacing Text 694
  - Changing the Look: Fonts and Colors 695
- PMEDIT 695
  - WPS Objects in PMEDIT 698
  - The Structural Elements of PMEDIT 700
  - Changing the Name of a Document 701

**Index of Listings** 703

**Index** 707

**How to Use the Disk** 719





# **OS/2 2.1 Workplace Shell Programming**



# An Introduction to Presentation Manager

The OS/2 operating system was introduced to the personal computer market at the end of 1987 with version 1.0. New features, distinct from those available in DOS, were to be found in its multitasking capabilities, the greater amount of memory available to each single running application (at that time up to 16MB of physical RAM in the system), and, above all, the adoption of a graphical user interface called *Presentation Manager* (PM) in place of the classic system prompt. Presentation Manager was released starting with version 1.1 of the operating system (December 1988). Then a second, more advanced version was released in September 1989 together with OS/2 1.2. The 16-bit version reached its final form in September 1990 with version 1.3. Despite its long-time availability, OS/2 has not seen any greater market acceptance—rather it has almost fallen into obscurity in consequence of the split between IBM and Microsoft, the two companies originally in charge of the system's design and development.

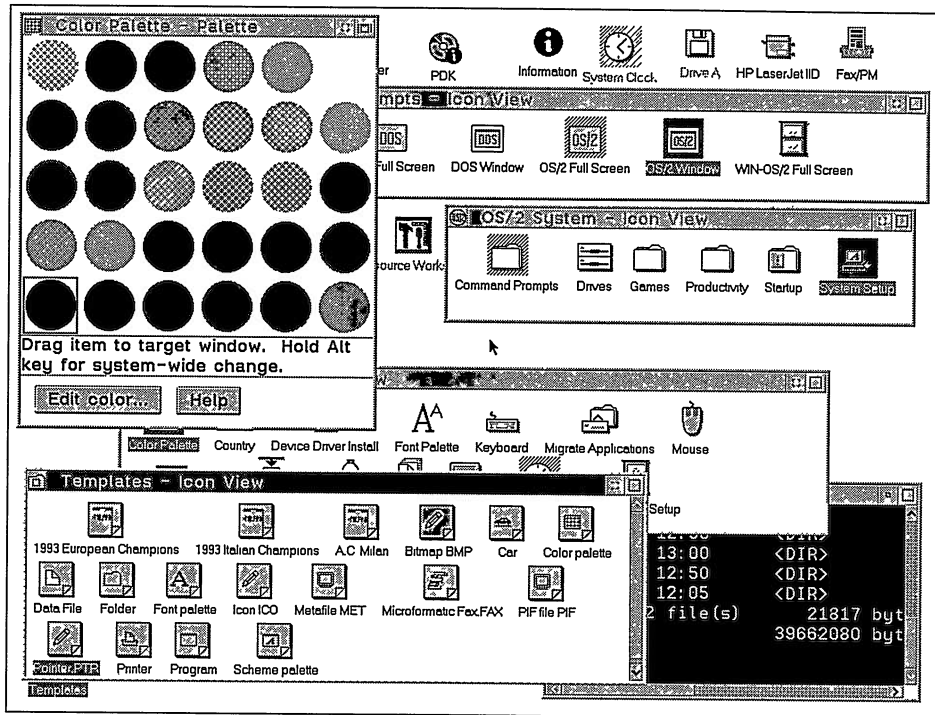
With the advent of version 2.0 in April 1992, OS/2 acquired yet more new features—completely developed by IBM (although there are still some signs of the former partnership with Microsoft), it is dedicated to Intel and closely compatible 32-bit processors. In June 1993, IBM released version 2.1.

This book discusses the structural and logical rules that govern the development of OS/2 2.1 applications running under the PM user interface, whereby each program is disguised as one or more graphical windows that can be displayed on the screen simultaneously (Figure 1.1). In OS/2 it is possible to develop applications that stand out from others both for their exterior, visual aspects as well as for their inner workings, strongly related to the executable code running on the microprocessor.

---

## OS/2 Applications

The OS/2 operating system takes full advantage of the 32-bit *protected* mode of Intel's iAPX86 processor family. This operating mode is available in the 80386, 80486, and *Pentium* processors, including all their SX, DX, and DX2 variants. The system can



**Figure 1.1** The aspect of OS/2 2.1 system is characterized by the simultaneous presence of several visible windows.

address up to 4GB of physical space, temporarily limited to 512MB for each 32-bit process. The rules for developing an OS/2 application differ significantly from those you might be accustomed to in DOS. These differences allow you to take full advantage of this new operating system's features, such as *InterProcess Communications*, flat memory model, semaphores, protected memory, and others. Some features are similar to those found in the MS Windows environment as far as the graphical user interface is concerned. The greatest advantage of OS/2 PM is its ability to exploit all the power of a fully preemptive multitasking system, providing better performance and a higher level of protection for each task in execution. The programming languages that can be used are those most commonly found on other hardware and software platforms, although OS/2 has a strong bias towards C and C++ for their flexibility and power.

The various applications available for OS/2 are commonly grouped into three distinct categories:

- Full-screen character-based kernel applications (FS)
- Character-based kernel applications runnable inside a graphical PM window (WIN)
- Presentation Manager applications (PM)

This list is extended with DOS, DOS-Extender, and MS Windows 2.x or 3.x applications. Compatibility with all these different 16-bit programs is granted by the support of virtual machines (VM), a feature typical of 32-bit processors. All these categories are not OS/2 2.x-specific applications, and therefore they cannot exploit all of the system's capabilities. We will limit our attention only to those executable modules that are specific to OS/2, that is, pure 32-bit code. Let's examine the OS/2 2.x native applications.

**Full-Screen Applications (FS).** This category of applications contains a great many programs that are mostly ports of previous DOS applications that have simply been recompiled for OS/2 (Figure 1.2).

The degree of adaptation of existing code to OS/2 varies considerably. It can be as shallow as a simple recompilation in order to render the executable application compatible with the new system loader. Or it can be as complex as a complete redesign to include and employ a great many new features available in OS/2, such as exploiting the greater memory address space.

The first applications ever marketed for OS/2 belonged to this category. The reason for this being the ease with which each software manufacturer could convert its products. FS applications are constructed by relying on a specific Application Programming Interface (API) of OS/2, which is essentially concentrated in the three subsystems KBD, VIO, and MOU, which handle respectively the keyboard, the video, and the mouse. This is still a 16-bit API in OS/2 2.x, and there is no plan for converting it to 32-bit. In this consideration there is an underlying strong message. A developer can still write FS applications, though he or she must not.

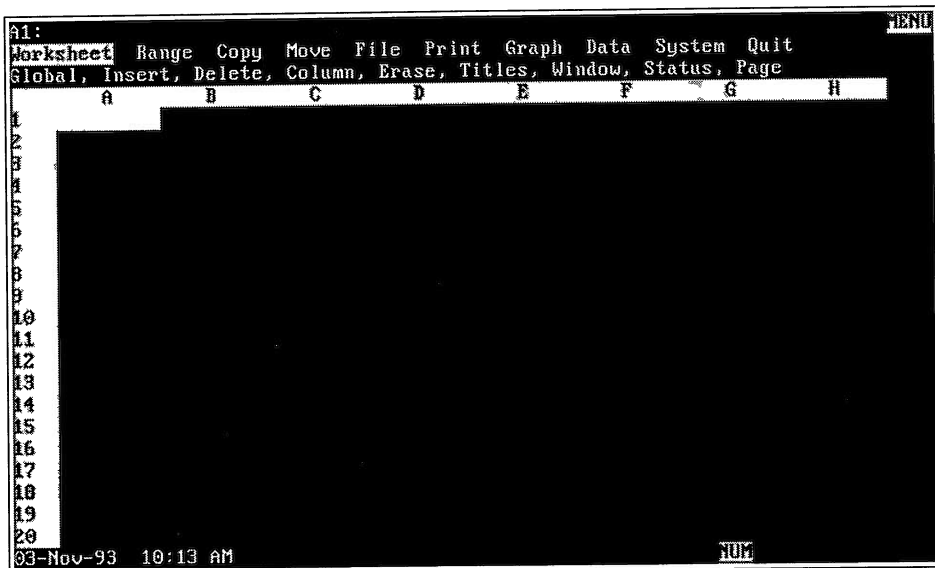


Figure 1.2 Lotus 1-2-3 is a typical representative of character-based user interface applications running under OS/2.

#### 4 OS/2 2.1 Workplace Shell Programming

**Windowed Applications (WIN).** Windowed applications differ from the previous ones only in that they are capable of running inside a PM graphical window. From the point of view of coding and programming, this means that the designer has to base all I/O activity on a specific set of services provided by the operating system (VIO subset), and avoid any I/O tools available in the language or in some function libraries (Figure 1.3).

A positive aspect of these programs is that they run just like graphical applications in PM, with all the advantages of ease of use. Despite the presence of a window that limits the output surface of similar applications, the program's output is always designed on the assumption that it has indiscriminate access to a full screen of 80 x 25 columns. This means that the diminished output surface of a WIN application does not imply any kind of reorganization of its output, but only a limit of its display. In order to show the whole output of a WIN application it is necessary to maximize its window to full screen size, and therefore run the application as if it were a FS application.

**PM Applications.** This is the category of the "true" OS/2 applications. The role played by Presentation Manager is, in fact, central to all of OS/2. The system's main interface is known as the *Workplace Shell (WPS)*, and it is the first application that takes

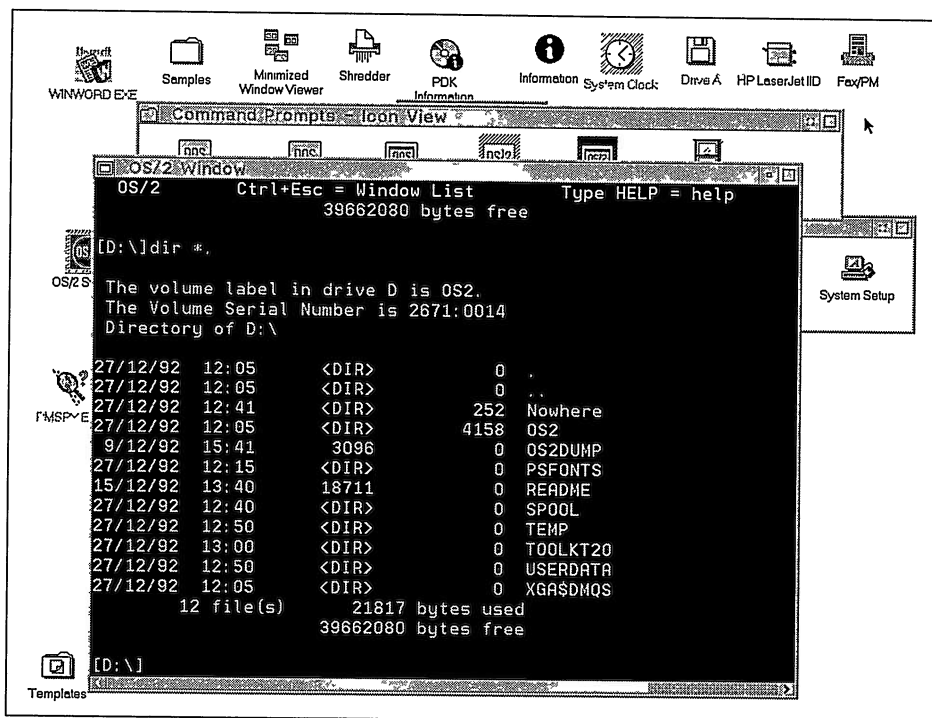


Figure 1.3 OS/2's system command processor, CMD.EXE, is a character based application that can be run in a graphical window.

full advantage of PM's API and of the objects of the System Object Module (SOM), a language-neutral environment for defining, managing, and interacting with class libraries. (Figure 1.1).

---

## Multitasking in OS/2

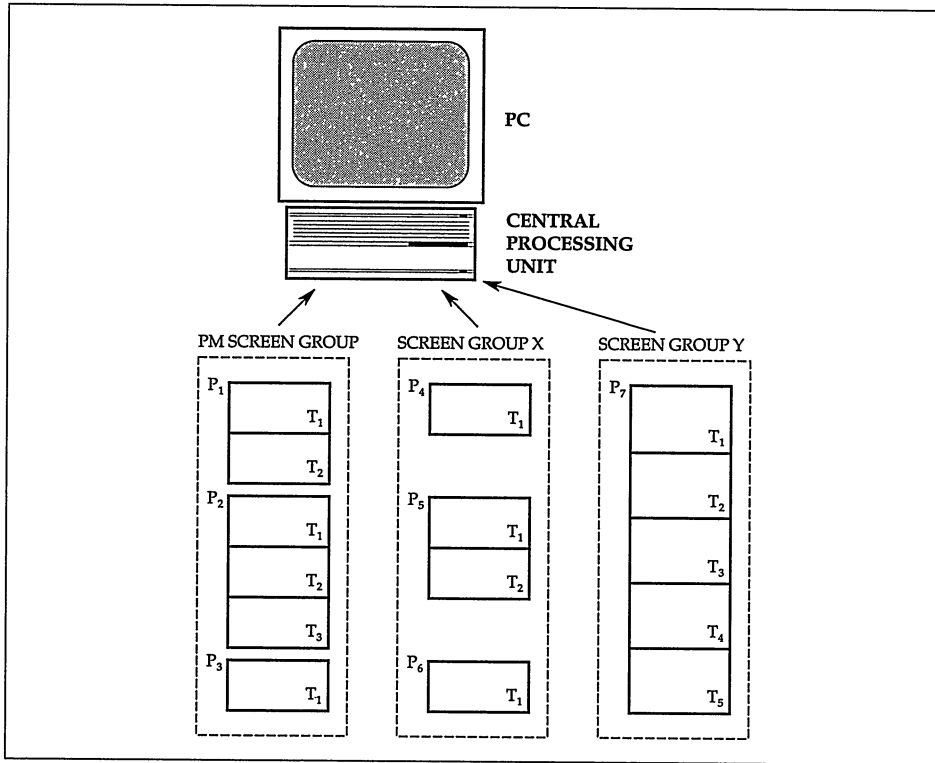
One of the distinctive features of OS/2 is its ability to run more than one application simultaneously. OS/2's multitasking is hardware supported, and is technically known as *preemptive multitasking*. This term means that there is a coordination mechanism that oversees the execution of code on part of the system's *scheduler*. In the configuration file, CONFIG.SYS, the *timeslice* parameter is used to express the minimum and maximum amount of time, in milliseconds, that a piece of code can keep running uninterruptedly on the CPU. Once this time is up, the program is taken off the processor, whether or not it has terminated the operations it was engaged in when preempted. When a program is preempted this way, its execution state is completely saved. The replacement of the code being executed can also take place before the *timeslice* actually expires; this will happen when an application with a higher priority is presented to the system. The *scheduler* identifies the new piece of code that needs to be transferred to the CPU by grouping applications in four priority classes, each one featuring an additional 32 internal levels of differentiation.

To respond to the requirements of multitasking, the system employs a shared logic of I/O tools (video, keyboard, and mouse). To this end, OS/2 virtualizes all I/O devices for a total of 16 application execution contexts (virtual PCs), also known as *screen groups* or sessions. Each session contains one or more running processes, and each process can be made up of one or more *threads*. Thus, the multitasking terms of OS/2 are the following:

- Screen groups
- Processes
- Threads

A screen group is a process container. Processes consist of one or more threads. By the term process we mean a running application with all of its associated resources—for example, memory blocks dynamically allocated during execution. The term *thread* indicates the minimum amount of code that can be addressed to the CPU by part of the scheduler. A thread in C coincides with a function. Figure 1.4 summarizes the relationships that exist among the multitasking elements of OS/2.

Not all 16 screen groups present in the system are available to the user. Only twelve of them are accessible, and they are characterized by a user interface controlled by the CMD.EXE command processor, the character user shell. One of them is dedicated to Presentation Manager, and four are reserved and/or perform tasks strictly related to the requirements of the previous 16-bit versions of the system. The screen group to which we will turn our attention will be the one containing PM.



**Figure 1.4** Scheme of multitasking in OS/2 systems.

The number of processes running simultaneously under OS/2 is not limited by the number of active screen groups, and cannot even be assessed on such a basis, because each session allows a variable number of applications to run. The system sets a limit to its multitasking capabilities only in terms of maximum number of simultaneously running threads, a limit equal to 4096. This is a purely theoretical number, a *non-limit*. You could think of just one application consisting of 4096 threads, or, at the other extreme, 4096 single-thread applications.

Since every application consists of at least one thread, although it will most often contain several threads (multi-threaded applications), the real number of simultaneously running processes in OS/2 is quite high—certainly much higher than the needs of a typical user, or even a power-user.

The relationship between the screen groups of character-based application and PM favors the former, since an FS program will take up the entire screen, while all PM applications have to share one (Figure 1.5).

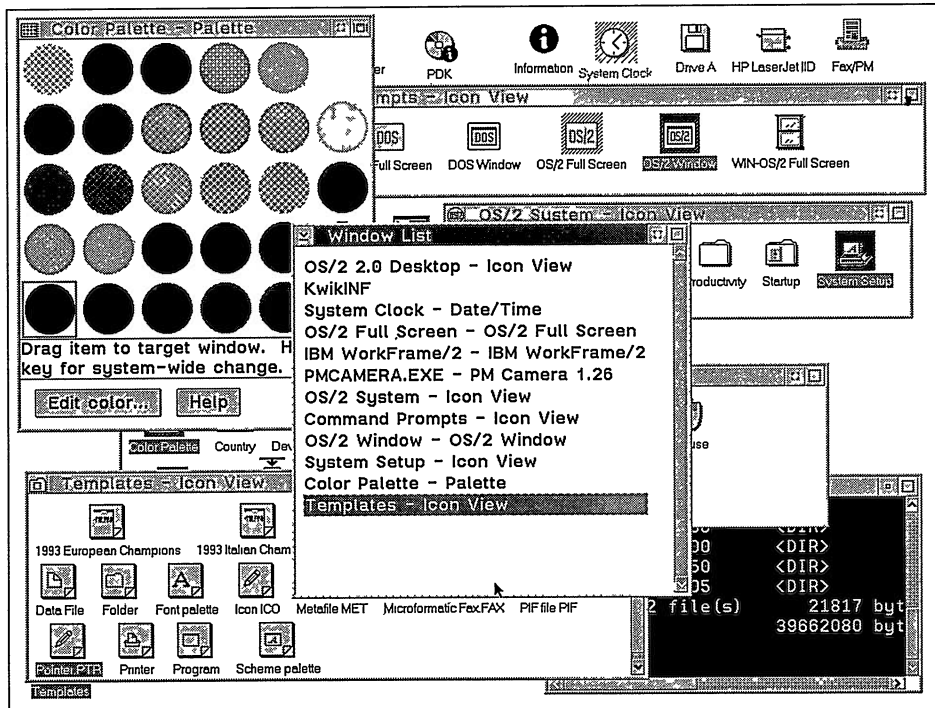
This means that the PM screen group allows several processes to be running at the same time. Actually, it will handle these processes according to rules that are different from those that govern all other screen groups. Figure 1.5 shows the subdivision of the system into screen groups, and shows the unique qualities of PM.



<i>Quantity</i>	<i>Type of Screen Group</i>
1	Presentation Manager
1	Reserved
1	Detached applications
1	Harderr
12	Full Screen

**Figure 1.5** Distribution of screen groups in OS/2 1.x.

The switching from screen group to screen group takes place according to a round-robin system. By pressing the keyboard combination ALT+ESC you can transfer to the display device of the system all active screen groups, one after the other, including the PM screen group. To directly access Window List, the simple PM application containing a list of all active tasks (either PM tasks or kernel processes), press CTRL+ESC (Figure 1.6).



**Figure 1.6** Window List containing the list of all applications, both PM and non-PM, that are active in the system.

## The PM Screen Group

The features of the PM screen group are not limited to the simultaneous execution of several applications sharing the screen's surface. It is also activated immediately after the preliminary *bootstrap* phase. Furthermore, the graphical interface of the system, WPS, is a PM application that is loaded immediately after bootstrap (Figure 1.7). Working in OS/2 is simple and intuitive thanks to WPS, a new breed of *object-oriented* user interface, a full replacement and enhancement of the interface of version 1.x.

### Working in the PM Screen Group

When the system is turned on, the user will be interacting directly with the PM screen group. The screen will look like the one shown in Figure 1.7.

Several icons will be displayed. As opposed to *application-oriented* interfaces like MS Windows or versions 1.x of OS/2, WPS is an *object-oriented* user interface. The icons that show up on the desktop can be broken into four categories:

- Folders
- Physical devices
- Data files
- Programs

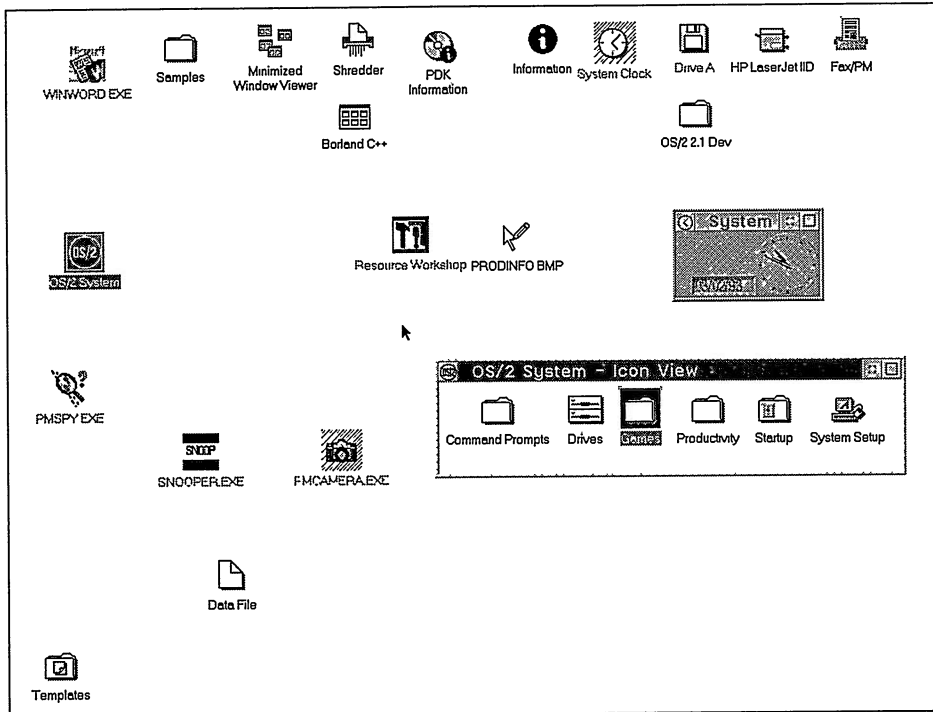
In previous versions, the appearance of icons was strongly related to running applications in their minimized state. In OS/2 WPS the scenario is different; the icons do not provide a visual cue. Double-clicking on an icon is most often followed by a window opening on the screen, as in the case of folders, the activation of an application (data files and programs), or the display of the settings of a device. Another interesting feature of WPS is that menus are not related to the traditional menu bar. Each WPS object is armed with a *window context menu* that is activated by pressing the right mouse button. In the next few chapters we will explore the relationship that exists between PM's API and these functional aspects of WPS, and discuss several examples that illustrate in greater detail how to create folders and window context menus. WPS itself is a PM application exploiting many features of the PM API, like predefined window classes.

---

## The API of OS/2

In order to proceed with the development of a PM program it is necessary to use certain functions of the operating system. The entire set of services offered by OS/2 is called API (*Application Program Interface*). The API of OS/2 can be classified according to their functions (Table 1.1).

As you can see, there are hundreds of calls available to the development process. The software designer will call the API services when immediate solutions are needed for certain problems related to the application's design. To open a file, call `DosOpen()`.



**Figure 1.7** An OS/2 system immediately after bootstrap operations.

*DosAllocMem()* is the easiest way to allocate a chunk of memory. The creation of a window is implemented through functions like *WinCreateWindow()*, *WinCreateStdWindow()*, *WinLoadDlg()* and others. The API of OS/2 is implemented like a call of the type `_System`, a calling convention defined by the designers of IBM, and is accessible and callable directly from any high-level language.

**Table 1.1** Prefixes That Characterize the Categories of API Functions of OS/2

<i>Prefix</i>	<i>Description</i>	<i>Quantity</i>
Win	Windowing	340
Gpi	Graphics	261
Drg	Drag&drop	32
Ddf	Data formatting	15
Prf	INI management	18
Prt	Printers	5
Dos	Control Program	152
Dev	Management of logical devices	5
Spl	Spooler	41

## 10 OS/2 2.1 Workplace Shell Programming

This approach to the API makes life a lot easier for programmers, as they are free to develop code independent of the language being used, and that is mostly based on tools offered by the system. Furthermore, calling the API directly will make the listing easier to read and understand.

The following code fragment is extracted from a typical OS/2 application that searches for a given file among all files in the system.

```
...
do
{
    // store the ID
    papprec -> lType = lType ;
    if( lType < 0L)
        papprec -> lType = j + inc ;

    // load string
    WinLoadString(HAB( hwnd), NULLHANDLE,
                 OBJECTID + papprec -> lType,
                 sizeof( szString), szString) ;
    psz = strdup( szString) ;

    // filling the RECORDCORE structure
    prec -> flRecordAttr = CRA_CURSORED | CRA_DROPONABLE ;
    prec -> pszIcon = psz ;
    prec -> hptrIcon = WinLoadPointer( HWND_DESKTOP,
                                     NULLHANDLE,
                                     OBJECTID +
                                     papprec -> lType) ;

    prec -> hptrMiniIcon = NULLHANDLE ;
    prec -> hbmBitmap = NULLHANDLE ;
    prec -> hbmMiniBitmap = NULLHANDLE ;
    prec -> pTreeItemDesc = NULL ;
    prec -> pszText = psz ;
    prec -> pszName = psz ;
    prec -> pszTree = psz ;
    // increase the counter
    j++ ;
} while( prec) ;
...
```

All functions that have the *Dos* prefix are API calls that can be invoked like any function of the C language, following their syntax. When developing applications for PM you will most often use the entire set of functions characterized by the prefix *Win*, and occasionally the functions with the prefix *Gpi*. Sometimes, listings will also contain calls to *Dos*, *Dev*, and *Prf*. In this book we focus almost exclusively on the API calls of the *Win* type—the set of system services that have been designed expressly for handling a window in PM.

## The Development of PM Applications

In order to develop a PM application, you must have a thorough understanding of the structural and functional features of OS/2. Unlike MS Windows, PM is the only user interface to the operating system to which all other problems (memory management, file I/O operations, process activation, creation of new threads, and so on) are delegated. Thus, a full study of the OS/2 kernel is beyond the scope of this book, and you should refer to the manuals that are furnished with the Toolkit for a thorough treatment. However, in some chapters, I will provide a brief description of memory management techniques, the use of semaphores, and the creation of threads.

### Software Tools

To write a PM application you must have already installed on your personal computer the OS/2 operating system, version 2.1 or even the older version 2.0, a high level language compiler that supports the 32-bit world of OS/2 and the Toolkit—the set of files necessary to create an executable starting from source modules. All examples in this book have been written based on the IBM development product suite: IBM WorkFrame/2, the IBM C/C++ compiler, and the IBM Toolkit. All listings also have been compiled and tested with the Borland C++ for OS/2 compiler, which is a valid alternative to IBM's product line. The hardware employed was an IBM model 90 486 66MHz DX2, equipped with 24 MB of RAM, two 400MB SCSI hard discs and an XGA video adapter. You don't need such a powerful system to use OS/2 or develop applications for it. Table 1.2 lists the extensions of some of the files that are necessary for creating PM applications that you will get acquainted with in the following chapters. The three main pieces of development software are the C compiler, the resource compiler, and the linker, listed in Table 1.3.

These tools are managed primarily by the development environment, WorkFrame/2, and are transparent to the developer.

**Table 1.2 File Extensions of the File Involved in the Development of a PM Application**

<i>Extension</i>	<i>Description</i>
.C	C source file
.H	Include or header source file
.DLG	File containing a dialog window template
.ICO	File containing an icon
.BMP	File containing a bitmap
.RC	File containing non-compiled resources
.OBJ	Object file produced by a compilation run
.RES	Compiled resource file
.DEF	Module definition file
.LIB	Collection of .OBJ files and import libraries
.EXE	Executable file produced by the linker

**Table 1.3 The Names of the IBM C Compiler**

<i>Module</i>	<i>Description</i>
RC.EXE	Resource compiler
ICC.EXE	Command for compiling and linking C programs
LINK386.EXE	Linker for producing 32-bit applications

## The C Language and PM

Almost all commercial PM applications are written in the C language, although it is possible to use other high-level languages like C++, COBOL, and Modula-2. The eventual next step is the transition to the C++ language, although that is not yet a viable solution because C++ products are still in the development phase. The main advantage of C++ over C is in the terseness of source code and in the power of the language's operators. Nevertheless, the tools furnished with the IBM C Set/2 do indeed make the development of programs easier.

### Header Files

Just like any application written in C, those for PM depend on some *header files*, all of which have the .H file extension. In the OS/2 Toolkit there is a tree structure of several header files that are often very large. In Appendix A, you will find a list of all the define arguments that are available in these header files and that are used in developing PM applications and to speed up the compilation phase. The main header file, for our purposes, is PMWIN.H, a file that is over 150K and contains the following:

- Prototype of the OS/2 API services with the *Win* prefix, that manage all problems involved in developing graphical windows
- Definitions of new simple and aggregate data types, used in the development of PM applications
- Simple defines and macros

In order to include in a piece of source code all elements specified in PMWIN.H, you need to specify the OS.H header file in an `#include` directive. However, before that you must write the directive `#define INCL_WIN` as in the following example:

```
#define INCL_WIN
#include <os2.h>
...
```

In this way the C source code will be able to access automatically all defines and API calls characterized by the prefix *Win*. In order to also have access to the *Gpi* portion of PM's API (described in PMGPL.H), you need only add a new preprocessor directive as in the following example.

```
#define INCL_WIN
#define INCL_GPI

#include <os2.h>
...
```

Both defines can be aggregated in an INCL\_PM directive:

```
#define INCL_PM
#include <os2.h>
...
```

Despite what you might think about the tree structure of the header files, it is necessary to specify the OS2.H header file rather than PMWIN.H directly, because some defines that affect the development of PM application are actually contained in other files, like OS2DEF.H.

The #define directives examined thus far affect only the access to the API that is typical of PM. However, in a PM application you might want to call *Dos*, *Dev* and even *Vio* functions. To meet this need, some specific directives allow the inclusion of other portions of the system's header files. Quite often, the first part of a source file for a generic PM application looks like this:

```
#define INCL_WIN    // whole API Win
#define INCL_GPI    // whole API Gpi
#define INCL_DOS    // whole API Dos
#define INCL_DEV    // whole API Dev

#include <os2.h>
...
```

This text is centered around the use of the *Win* functions, and thus, in addition to reading the on-line help text, you might also want to have a look at PMWIN.H and PMSTDDL.G.H—two excellent starting points for getting acquainted with PM's API. The growing complexity of the interface of OS/2 and the constant addition of new functions (for example, WPS), has inspired IBM developers to extend the contents of PMWIN.H and even add new header files. Among these, PMSTDDL.G.H and PMWPS.H play a fundamental role in the addition of new predefined window classes and to the new support of WPS.

## Handles

Among the various types defined through the keyword typedef of the C Language, *handles* play an important role in the *object-oriented* programming model of PM. Actually, handle is just a term that signifies a simple unsigned long (ULONG), a four byte datum through which the system can refer to complex objects like a window, a bitmap, a presentation space, or a device context.

```
typedef unsigned long LHANDLE;
```

Handles are used everywhere, and there are several kinds which differ only in naming convention: It is precisely for this reason that they make listings easier to read (Tables 1.4, 1.5, 1.6).

**Table 1.4 The Handles Declared in PMWIN.H**

<i>Handle</i>	<i>Description</i>
HENUM	Handle for enumeration operations.
HIMAGE	Handle of an image.
HSAVEWP	Handle for operations saving the position of a window.
HACCEL	Handle for an accelerator table.
HPOINTER	Handle for the mouse pointer.
HATOMTBL	Handle for a table of atoms.

**Table 1.5 The Handles Declared in OS2DEF.H**

<i>Handle</i>	<i>Description</i>
HMODULE	Handle of a module (DLL).
HAB	Handle to the application's anchor block.
HPS	Handle to a presentation space.
HDC	Handle to a device context.
HRGN	Handle to a region (surface that cannot be reduced to a standard geometrical shape).
HBITMAP	Handle to a bitmap.
HMF	Handle to a metafile.
HPAL	Handle to a palette.
HWND	Handle to a window.
HMQ	Handle to the message queue.

**Table 1.6 The Handles Declared in PMSHL.H**

<i>Handle</i>	<i>Description</i>
HPROGRAM	Handle to a program.
HAPP	Handle to an application.
HINI	Handle to a profile file.
HSWITCH	Handle to the list of applications in the Window List.



Almost all of the handles listed in Tables 1.4 through 1.6 will be presented in the examples in this book. The choice of handles as tools for controlling large objects (like bitmaps) is sound because they are easy to use and small in size. Most often, in PM listings you will find the `define NULLHANDLE`; it's the generic null handle:

```
#define NULLHANDLE    ( (LHANDLE)0)
```

Why do we have to deal with handles? The answer is a consequence of the development model. Every PM object (a window, a bitmap) is a collection of data stored somewhere in the system memory. Its location may vary during execution time due to the memory management techniques inside the processor and the operating system itself. In fact, optimization of memory usage requires that objects can be moved freely to reduce fragmentation. The address of each object varies from time to time; therefore, a pointer is not the most suitable way to access any memory location, since the operating system would have to constantly update the pointer within applications. Instead, when an application requests memory from the operating system, it is returned a handle, a dummy number which corresponds to a pointer to the actual address. Memory locations are handled by the system, handles by the application. You will never find `handle` as the *r-value* in an assignment operation, except when the *l-value* is another handle. Usually handles are treated as *l-values* where the *r-value* is an API call.

```
hwnd = WinCreateStdWindow( ... ) ;
hptr = WinLoadPointer( ... ) ;
hbmp = GpiLoadBitmap( ... ) ;
```

This demonstrates that handles are returned by the operating system and that you cannot make any assumptions regarding their numeric values—what really matters is to check if a handle is valid or not valid. Valid handles have positive values (greater than zero because it is a `ULONG` datum), while an invalid handle is zero, like `NULLHANDLE`.

## Data Types and the Defines in PM

Starting with the next chapter we will develop a true PM application by using the API environment. It will be necessary to use data types and defines that are set up in the OS/2 header files, so at this point we will examine the nature of PM's API in greater detail.

In the first place, the API functions follow the `_System` calling convention. These do not perform as well as the `_Optlink` adopted by the C Set++ compiler, but better respond to the needs of the API. The `_System` calling convention passes all parameters on the stack, just like the C calling convention, with a double word alignment. Unlike the C calling convention, though, the parameters are pushed onto the stack from right to left. The number of parameters is constant, like the *Pascal Calling Convention*. Their disposal after a call must be taken care of by the calling code. In the header files, `_System` will originate a `define` for an `APIENTRY` call:

```
#define APIENTRY    _System
#define EXPENTRY    _System
```

The same value is also assigned to the define EXPENTRY, used to mark all functions of this kind explicitly written in the source code by the programmer that are not part of the API of OS/2. The define EXPENTRY exports the corresponding function so that the executable module can be generated correctly by the linker (we will get back to this issue when dealing with the subject of the module definition file). OS2DEF.H also contains other definitions that are frequently used in PM programming (Table 1.7).

**Table 1.7 Some Frequently Used Data Types in PM Programming**

<i>Frequently Used Data Types in PM</i>	<i>Prefix/Description</i>
#define VOID void typedef unsigned long APIRET;	Value returned by an API call of the DOS type
#define FALSE 0 #define TRUE 1	
#define NULL ((void *)0)	NULL POINTER
#define NULLHANDLE ( (LHANDLE) 0)	NULL HANDLE
#define CHAR char	ch
#define SHORT short	s
#define LONG long	l
#define INT int	i
typedef unsigned char UCHAR;	uch
typedef unsigned short USHORT;	us
typedef unsigned long ULONG;	ul
typedef unsigned int UINT;	ui
typedef unsigned char BYTE;	b
typedef unsigned char *PSZ;	psz
typedef unsigned char *NPSZ;	
typedef unsigned char *PCH;	pch
typedef unsigned char *NPCH;	
typedef BYTE *PBYTE;	pb
typedef BYTE *NPBYTE;	
typedef CHAR *PCHAR;	pch
typedef SHORT *PSHORT;	ps
typedef LONG *PLONG;	pl
typedef INT *PINT;	pi
typedef UCHAR *PUCHAR;	puch
typedef USHORT *PUSHORT;	pus

(continued)

**Table 1.7 (Continued)**

<i>Frequently Used Data Types in PM</i>	<i>Prefix/Description</i>
<code>typedef ULONG *PULONG;</code>	pul
<code>typedef UINT *PUINT;</code>	pui
<code>typedef VOID *PVOID;</code>	pv
<code>typedef PVOID *PPVOID;</code>	ppv
<code>typedef unsigned long BOOL;</code>	f
<code>typedef BOOL *PBOOL;</code>	pf

The values expressed in the second column indicate what prefix needs to be used to mark each identifier for that particular data type in order to reflect the Hungarian Notation conventionally used by PM programmers. *Hungarian Notation* is a convention that simplifies software maintenance and readability. It was invented by Charles Simonyi, a Hungarian software developer. The notation relies upon the presence of a lowercase letter prefix that identifies the data type of an identifier (for example, `psz` to indicate a pointer to an ASCIIZ string), and one or more full words with initial capital letters. For example, if you need to declare an unsigned long to store the description flags of a window, you might come up with an identifier like `ulFrameControlFlags`. This might be further abbreviated to `ulFCFlags`, provided its usage context is always unambiguous.

If you want to declare an unsigned short, you can specify the data type `USHORT` and prefix the identifier with `us` as is shown in Table 1.7. A data type independent array can be introduced by the letters `rg` (range); an array of pointers to the char data type will thus be called something like `rgszStrings[]` or even `szString[]`.

Appendix B contains a detailed description of the Hungarian Notation.

## ***The EXPENTRY Functions***

As we will see in the next chapter, the functions that are characterized by the define `EXPENTRY` always return a value of type `MRESULT` and take two parameters of the type `MPARAM` as defined in `PMWIN.H` (Table 1.8).

**Table 1.8 The Parameters and Return Values of an EXPENTRY Function Are Always of the Type MPARAM and MRESULT, Respectively**

<i>Parameters</i>	<i>Prefix</i>	<i>Description</i>
<code>typedef VOID *MPARAM;</code>	mp	Parameter to a window procedure
<code>typedef MPARAM *MPPARAM;</code>	pmp	Pointer to a MPARAM
<code>typedef VOID *MRESULT;</code>	mres	Return value of a window procedure
<code>typedef MRESULT *PMRESULT;</code>	pmres	Pointer to a MRESULT

In each traditional PM application, there will always exist at least one function declared as EXPENTRY, and thus it is important to know its characteristics and function. Furthermore, some macros defined in OS2DEF.H and in PMWIN.H play a fundamental role in making PM programming easier through functions of the EXPENTRY type (Tables 1.9 and 1.10).

**Table 1.9 List of Macros Used by Functions That Have the EXPENTRY Modifier**

<i>Macro / Description</i>
MAKETYPE(v, type) *((type *)&v) Casts an identifier into any other data type.
FIELDOFFSET(type, field) ((SHORT)&(((type *)0)->field)) Computes the offset of any kind of data structure.
MAKEULONG(l, h) ((ULONG)(((USHORT)l)   ((ULONG)((USHORT)h)) << 16)) Builds a ULONG starting with two SHORTs.
MAKELONG(l, h) ((LONG)MAKEULONG(l, h)) Builds a LONG starting with two SHORTs.
MAKEUSHORT(l, h) (((USHORT)l)   ((USHORT)h) << 8) Combines two 8 bit objects to create a USHORT.
MAKESHORT(l, h) ((SHORT)MAKEUSHORT(l, h)) Combines two 8 bit objects to create a SHORT.
LOBYTE(w) LOUCHAR(w) Extracts the lower byte.
HIBYTE(w) HIUCHAR(w) Extracts the higher byte.
LOUCHAR(w) ((UCHAR)(w)) Extracts the lower UCHAR.
HIUCHAR(w) ((UCHAR)(((USHORT)(w) >> 8) & 0xff)) Extracts the higher UCHAR.
LOUSHORT(l) ((USHORT)((ULONG)l)) Extracts the lower USHORT.
HIUSHORT(l) ((USHORT)(((ULONG)l) >> 16) & 0xffff)) Extracts the higher USHORT.

**Table 1.10 Extraction of Data from a MPARAM**

*Creation of a MPARAM Starting with Standard Data Types*

MPVOID ((MPARAM)0L)  
MPFROMP(p) ((MPARAM)((ULONG)(p)))  
MPFROMHWNDR(hwnd) ((MPARAM)(HWNDR)(hwnd))  
MPFROMCHAR(ch) ((MPARAM)(USHORT)(ch))

(continued)

**Table 1.10 (Continued)**

---

**Creation of a MPARAM Starting with Standard Data Types**

---

MPFROMSHORT(s) ((MPARAM)(USHORT)(s))  
 MPFROM2SHORT(s1, s2) ((MPARAM)MAKELONG(s1, s2))  
 MPFROMSH2CH(s, uch1, uch2) ((MPARAM)MAKELONG(s, MAKESHORT(uch1, uch2)))  
 MPFROMLONG(l) ((MPARAM)(ULONG)(l))

---

**Macros to Extract Standard Data Types from a MPARAM**

---

PVOIDFROMMP(mp) ((VOID\*)(mp))  
 HWNDFROMMP(mp) ((HWND)(mp))  
 CHAR1FROMMP(mp) ((UCHAR)(mp))  
 CHAR2FROMMP(mp) ((UCHAR)((ULONG)mp >> 8))  
 CHAR3FROMMP(mp) ((UCHAR)((ULONG)mp >> 16))  
 CHAR4FROMMP(mp) ((UCHAR)((ULONG)mp >> 24))  
 SHORT1FROMMP(mp) ((USHORT)(ULONG)(mp))  
 SHORT2FROMMP(mp) ((USHORT)((ULONG)mp >> 16))  
 LONGFROMMP(mp) ((ULONG)(mp))

---

**Macros to Create a MRESULT from Standard Data Types**

---

MRFROMP(p) ((MRESULT)(VOID\*)(p))  
 MRFROMSHORT(s) ((MRESULT)(USHORT)(s))  
 MRFROM2SHORT(s1, s2) ((MRESULT)MAKELONG(s1, s2))  
 MRFROMLONG(l) ((MRESULT)(ULONG)(l))

---

**Macros to Extract Standard Data Types from a MRESULT**

---

PVOIDFROMMR(mr) ((VOID\*)(mr))  
 SHORT1FROMMR(mr) ((USHORT)((ULONG)mr))  
 SHORT2FROMMR(mr) ((USHORT)((ULONG)mr >> 16))  
 LONGFROMMR(mr) ((ULONG)(mr))

---



---

## Debugging in OS/2 PM

An important element by which you can judge the maturity of a programming environment is by the quantity and quality of debugging tools. Some people consider debuggers tools you cannot live without, others think they are just cute accessories. Nonetheless, you will certainly feel more relaxed in knowing that you can go hunting bugs with tools like the IBM PM Debugger, which is bundled with the IBM C Set++ compiler. With this debugger you can easily and quickly inspect any executable module (EXE) while it is running. One of the manuals of the IBM C compiler is devoted

to IPMD, and presents detailed information on all facets of debugging—even in a multithreaded environment like that of OS/2 PM. In this text we use IPMD both to catch any bugs that might have crept into the code, as well as using it as a simple learning tool. All examples in this text have in their corresponding WorkFrame/2 project files the debug flags set both for compiling as well as for linking.

## Kwikinf

All IBM development tools are supported by rich on-line documentation, full of cross-references that make it a breeze to use. The best way to access the on-line help is by pressing the keyboard combination ALT+Q, which will bring the KWIKINF.EXE application on screen (Figure 1.8).

The installation of the toolkit will transfer KWIKINF.EXE and its related files to the TOOLK21\OS2BIN directory. It is a good idea to place this file in the *Startup* folder, so that it will run automatically each time you turn on the system.

When using the advanced editor, EPM.EXE, you can access the on-line documentation by pressing the keyboard combination CTRL+H. You can also request help regarding a message or function by pressing that keyboard combination when the cursor is above an API item.

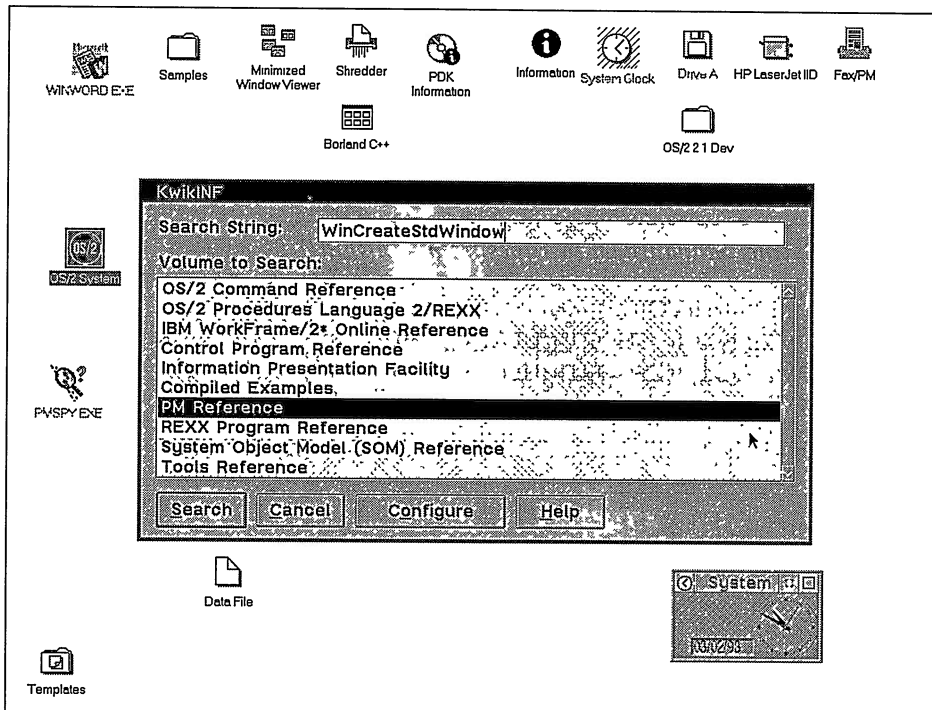


Figure 1.8 KWIKINF makes the software developer's life easier by presenting on-line the whole of the development toolkit's documentation.

# The Development Model

The completion of a PM application follows a specific logical path based on a precise development model. Differences as compared to an OS/2 kernel program (i.e., character based applications) are to be found both in the number and type of files involved. A PM application is clearly distinguished on the screen by one or more windows. (We will use the term *generic PM application* to refer to any program that shows up just in an on-screen window.) The number of windows and the specific type of relationship established among them is strongly affected by the interface model adopted by the developer. The style guidelines of the *Common User Access 91* (CUA 91) recommend a *Single Document Interface* (SDI) approach rather than the previously preferred *Multiple Document Interface* (MDI) adopted in OS/2 1.3 and derived from MS Windows 3.x. Adhering to the CUA 91 specifications rather than the former CUA 89 specifications involves a significant impact even on the software developer's work. We will stress the stylistic guidelines and aspects to be followed whenever there are discrepancies between the previous interface model and the new one.

In order to create and display a PM window/application, you must write several files. On the whole, due to the specific rules of the PM development model, the effort required is substantially greater than that required by a *full screen* program. Before examining any code, let's define some structural elements and the terminology used by a typical PM application (Figure 2.1).

It's easy to recognize many of the structural elements just by looking at a PM window. Each and every PM window has a border that delimits the area taken up on the screen. This border can have different attributes and aspects, depending on what has been set up in the original source code. In the case of Figure 2.1, you can see a typical sizing border, which allows you to change the size of a window simply by pointing the mouse on one of the four borders, depressing the left mouse button, and, while keeping that mouse button down, moving the mouse pointer on the desktop. When you release the mouse button, the window will be resized according to what has been indicated by the user. (There is also an equivalent procedure for keyboard users.)

```

E.EXE - d:\config.sys
File Edit Options Help
SET VIO_XGA=DEVICE(BVHVGA,BVHXGA)
SET VIDEO_DEVICES=VIO_XGA
DEVICE=D:\OS2\XGARING0.SYS
IFS=D:\OS2\HPFS.IFS /CACHE:1024 /CRECL:4 /AUTOCHECK:CDEF
PROTSHELL=D:\OS2\PMShell.EXE
SET USER_INI=D:\OS2\OS2.INI
SET SYSTEM_INI=D:\OS2\OS2SYS.INI
SET OS2_SHELL=D:\OS2\CMD.EXE
SET AUTOSTART=PROGRAMS,TASKLIST,FOLDERS,CONNECTIONS
SET RUNWORKPLACE=D:\OS2\PMShell.EXE
SET COMSPEC=D:\OS2\CMD.EXE
LIBPATH=C:\CCMAIL;D:\OS2\DLL;D:\OS2\MDOS;D:\;D:\OS2\APPS\DLL;F:\TOOLKT21\DLL;F:\IBMWF\
DLL;F:\IBMCPD\DLL;C:\MMOS2\DLL;E:\UTS\UTL;
SET
PATH=D:\OS2;D:\OS2\SYSTEM;D:\OS2\MDOS\WINOS2;D:\OS2\INSTALL;D:\;D:\OS2\MDOS;D:\OS2\AP
PS;F:\TOOLKT21\OS2BIN;F:\IBMWF\BIN;F:\IBMCPD\BIN;C:\MMOS2;E:\UTS\UTL;
SET
DPATH=D:\OS2;D:\OS2\SYSTEM;D:\OS2\MDOS\WINOS2;D:\OS2\INSTALL;D:\;D:\OS2\BITMAP;D:\OS2\
MDOS;D:\OS2\APPS;F:\TOOLKT21\BOOK;F:\IBMCPD\LOCALE;F:\IBMCPD\HELP;F:\IBMCPD\SYS;C:\MM
OS2;C:\MMOS2\INSTALL;E:\UTS\UTL;E:\UTS\UTD;
SET PROMPT=$|I$P|
SET
HELP=D:\OS2\HELP;D:\OS2\HELP\TUTORIAL;F:\TOOLKT21\OS2HELP;F:\IBMWF\HELP;F:\IBMCPD\HEL
P;C:\MMOS2\HELP;E:\UTS\UTL;
SET GLOSSARY=D:\OS2\HELP\GLOSS;
SET IPF_KEYS=SBCS
PRIORITY_DISK_IO=YES

SET UMPATH=E:\UTS\UTL;E:\UTS\UTD;

File Edit Options Help
  
```

Figure 2.1 The OS/2 System Editor is a typical PM application accommodating a menu bar, a vertical scrollbar, and resizing buttons.

The darker border (which appears dark green in the default PM coloring scheme) occupies the upper portion of the window shown in Figure 2.1. In this area, there is a left-aligned text item that corresponds to the program's title and identifies the window/application. This is known as the *titlebar*, and is the distinguishing feature between the windows of the same or different applications. The color of the titlebar is varied by the system according to the window's status. If the window is active the default color is green; if the window is inactive, the color is gray.

To the left of the titlebar there is an icon (a bitmap) distinguished by a small arrow pointing downward and a small horizontal bar. In OS/2 2.1 systems, this image is often replaced by a colored bitmap corresponding to the application's own icon. To understand this description more clearly, just open the *OS/2 System* object. If you have an XGA video adapter, the bitmap will show up as a small circle with "OS/2" printed inside. With lower resolution video systems, the image is limited to the circle alone. Besides the actual appearance of the *titlebar icon* (this is its real name, even though you might be tempted to call it the system menu icon, as in the past), by selecting it with the keyboard or the mouse, you will see a *drop-down menu*. By default, this menu presents eight options (Figure 2.2). Here we need to distinguish between the CUA 89 and the CUA 91 models.



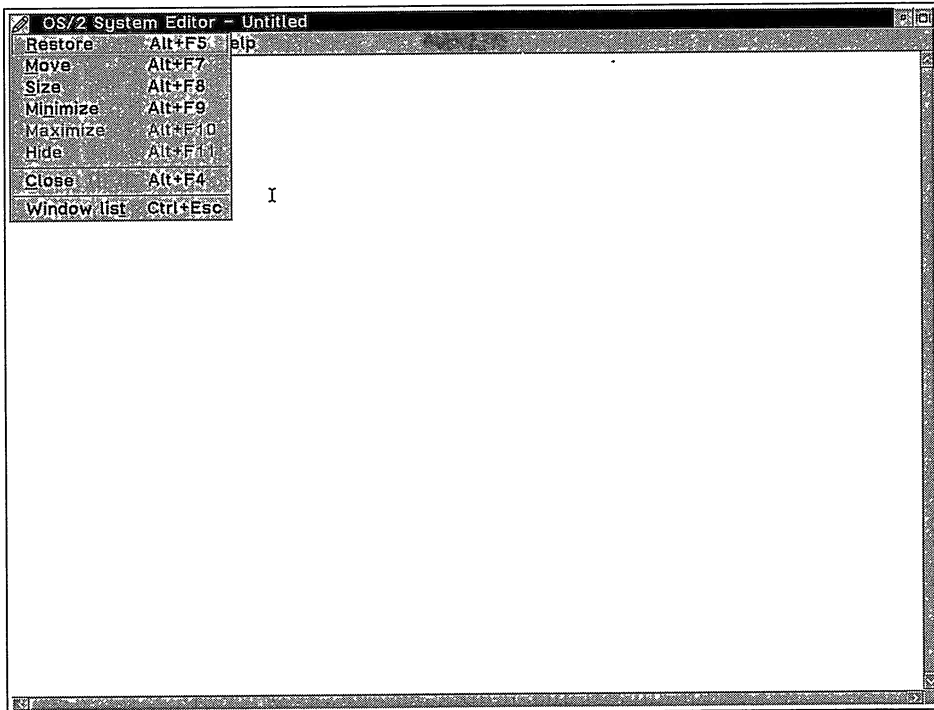


Figure 2.2 The structure of a default system menu in PM.

### *CUA 89 and the System Menu*

The icon that appears to the left of the titlebar is known as the *system menu*. It is actually a drop-down menu whose contents are consistent from application to application. This menu is designed to provide easy windows management even for users without a mouse. Thanks to the system menu it is possible to manipulate windows directly by means of keyboard operations. You can change a window's size (Size, Restore, Minimize, Maximize, and Hide), its on-screen position (Move)—you can even terminate the application running in the window (Close). You can also access the Window List, a special window listing all tasks that are active in OS/2 at any time.

### *CUA 91 and the System Menu*

CUA 91, on the other hand, assumes that two icons appear to the left of the titlebar: Of these two icons, the leftmost corresponds to the system menu. The drop-down menu, displayed by pressing Alt+Spacebar or Shift+Tab, contains the same options to control the overall size and position of the windows. However, in OS/2 2.1 there is no application whatsoever with a double icon to the left of the titlebar. The designers of WPS decided to merge the two menus corresponding to the two icons into one single menu. To be completely accurate, one should distinguish between three stylistic

approaches: CUA 89, CUA 91, and WPS. Chapter 6 will discuss the development of menus according to these diverse approaches.

## The Sizing Icons

To the right of the titlebar there are two icons, one featuring a very small square, and the other a somewhat larger square. Both images can be replaced under certain functions of the window, or as a result of some choice at the design level. The icon that appears nearest to the titlebar looks like a small square, of which only the corners are visible. The second icon looks like a window with two vertical bars along the sides. It is impossible to display all four icons simultaneously in a PM window.

The only way to operate these icons is to use the mouse. The action of any of these icons is summarized in the following table.

<i>Icon</i>	<i>Action</i>
Minimize	The window shrinks down to the size and shape of an icon inside the Minimize Window Viewer folder or on the desktop (and in this case it is also bordered by a frame).
Maximize	The window expands to cover the whole screen.
Hide	The window disappears from the screen.
Restore	The window regains the position and size it had before the last minimize or maximize operation.

The selection of any one of these icons automatically updates the contents of the window's system menu.

## The Menu Bar

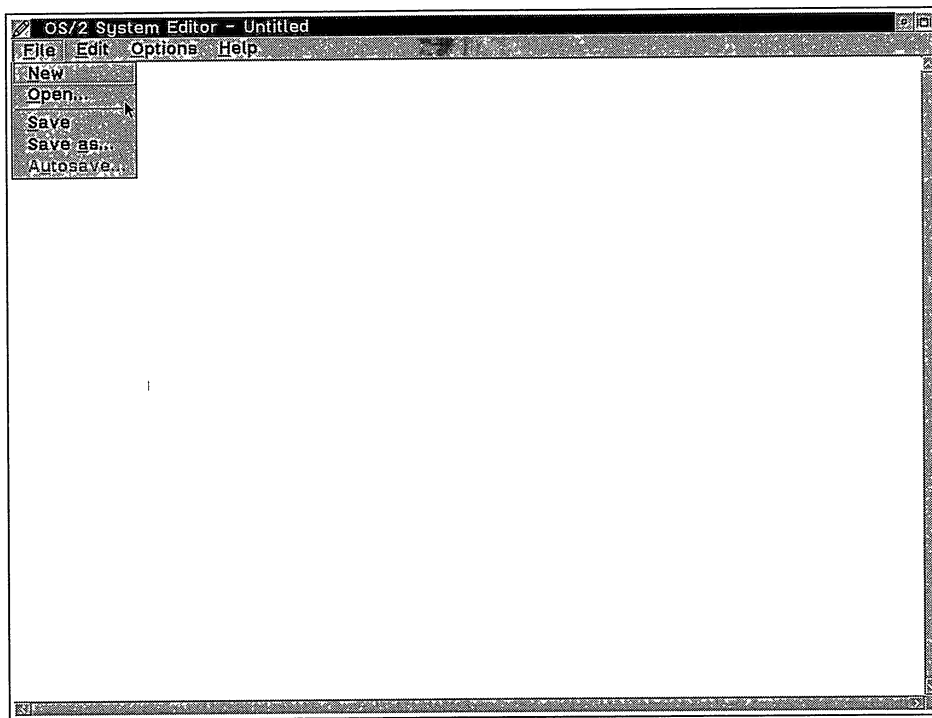
Immediately underneath the titlebar (Figure 2.1) is the *menu bar* (sometimes referred to as the *action bar*) which lists one or more *top-level menus*. By selecting the name of a top-level menu, you will make appear the corresponding drop-down menu, as you can see in Figure 2.3.

The portion of window enclosed by the right, left, and lower sizing borders, and by the menu bar from above, is known as the *client window*. The client window corresponds to the area within which an application can perform its output actions on the screen by displaying text, drawings, and images. In Figure 2.1 the client window is simply the blank portion of the window.

Now that we have described the basic elements of a PM window (although we will get back to this subject in Chapter 4), we will now examine the development model of a PM application.

## Flow Chart

Because of the many variables involved, it is impossible to determine the exact number of files needed to create a PM application. We will start to explore by looking at the development of a generic PM application. By this we mean a typical program that



**Figure 2.3** The File drop-down of the System Editor of OS/2.

creates and displays a window (and possibly other subordinate windows connected to the main window). With this in mind, the flow chart in Figure 2.4 is a good example, and demonstrates that six is the minimum number of files involved in the creation of a typical PM application.

In Chapter 1 we stressed the need for a header file for any PM application, and we have identified the model in OS2.H. It is also necessary to have a source module (a file with the extension C) containing the code that has to be compiled into an executable module. Finally, it is necessary to have a module definition file (with the extension DEF) to be fed to the linker. This file informs the linker about the functional features of the executable module it is about to create. A DEF file is optional for smaller PM programs, but it is mandatory for a full-blown PM application or DLL.

The development model usually requires the presence of a specific file containing resources that can be called upon and used on demand by the application during its execution. This kind of file is recognizable by the extension RC.

To complete the list of files involved in the creation of a PM application, we must not forget the *make file* (which has the extension MAK). A make file is simply a text file listing a series of directives that affect the building of the executable module, starting with the original source code file and the *project file* (PRJ). Thus, a typical PM program is constructed with at least the following six file types: PRJ, MAK, RC, DEF, H, and C.

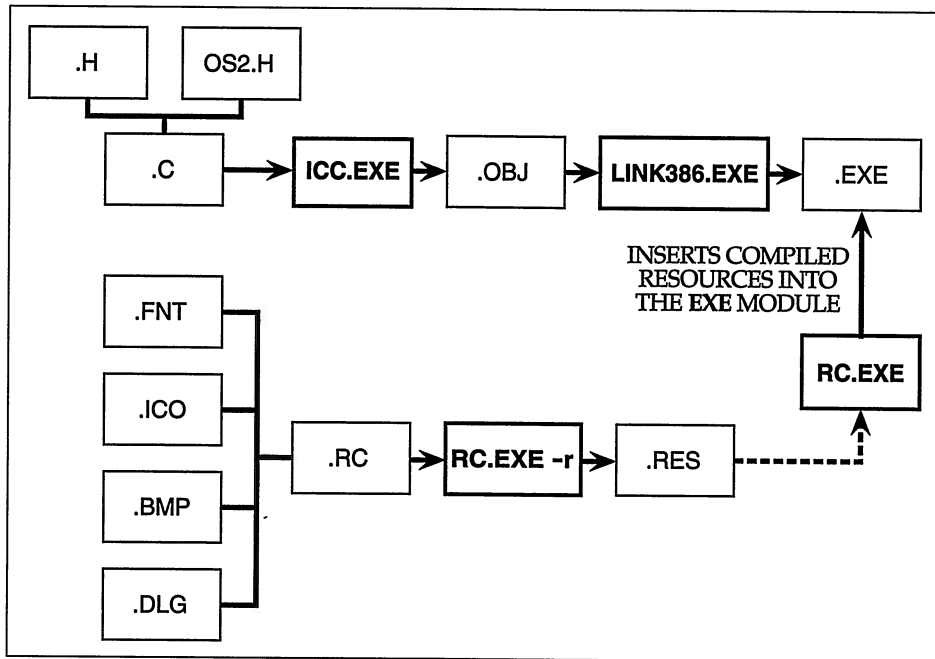


Figure 2.4 Flow chart for the development of a PM application.

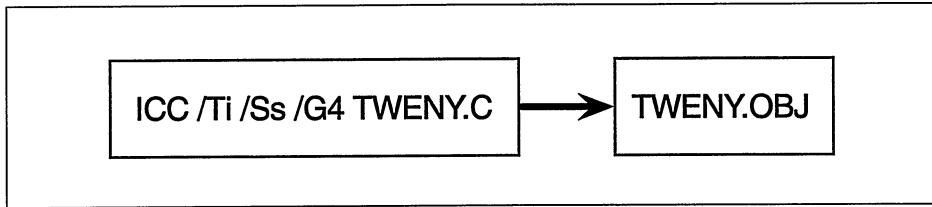
## The Make File

The construction of a complex program requires the repeated execution of the C language compiler, the building of traditional code libraries (with static linkage managed by the utility library LIB.EXE) or dynamic link libraries, the use of the linker and the resource compiler (Figure 2.4).

During the development phase, the compiler is used frequently and requires a great deal of time—easily exceeding several hours a day. One way to minimize the time required is to define a make file; that is, to construct a formal syntax of the various steps that are needed to build the executable module.

The most useful feature of a make file is that it tests each and every intervention request on part of the compiler, the linker, and even other utility programs. The tests verify the existence of the relationship among the files that make up the input of the utility programs (like ICC.EXE or LINK386.EXE) and their respective output files. The C compiler (ICC.EXE) needs to access a source file (C) in order to produce an object module (OBJ), as you can see in Figure 2.5.

The make file controls the execution of this kind of operation on the basis of the time stamp (date and time) of the source file (in this case the C source code) compared to the time stamp of the target file (OBJ). The actions will be undertaken only if the target file has an earlier time stamp than the source file.



**Figure 2.5** Transformation of the source code into an object module.

The contents of a make file can be divided into two categories of directives: setting directives and execution directives. Setting directives contain instructions that set the conditions for testing every time the make file is executed; to the second group of directives belong the operations that have to be executed in order to achieve the desired result. Listing 2.1 represents a simple make file for the development of a generic PM application.

**Listing 2.1 A Generic Make File Constructed According to the Traditional Scheme Adopted from the Very First Version of OS/2**

---

```

#-----
# MACHINE make file
#-----

machine.obj: machine.c
    cl -c -G2sw -W3 -Zipe -Od machine.c

machine.exe: machine.obj machine.def
    link machine, /CO /align:16, NUL, os2, machine
  
```

A make file usually is much more complex than what has been discussed above; the C compiler's documentation contains a detailed description of all capabilities of the NMAKE.EXE utility.

With the release of the ICC compiler, the role of make files is even more important, despite the fact that the compiler itself is now able to define the dependencies between the various modules that make up a program and create a DEP file. Operating inside WorkFrame/2, you can take advantage of the environment's ability to generate a make file, starting with the definition of the executable module you want to build with respect to the various source files that have been declared. Listing 2.2 shows a make file generated with the new syntax introduced by the NMAKE.EXE utility of the ICC compiler.

**Listing 2.2 A Make File Generated Directly from within the WorkFrame/2 Environment and Valid for the NMAKE.EXE Utility**

---

```

PROJ = MACHINE
PROJFILE = MACHINE.MAK
DEBUG = 1
  
```

```

PWBMAKE = pwbrmake
NMAKEBSC1 = set
NMAKEBSC2 = nmake
LINKER = link
ILINK = ilink
LRF = echo > NUL
BIND = bind
RC = rc
IMPLIB = implib
LFLAGS_G = /NOI /ST:8192/BATCH
LFLAGS_D = /CO /INC /F /PACKC /PACKD /PMTYPE:PM
LFLAGS_R = /E /F /PACKC /PACKD /PMTYPE:PM
MAPFILE_D = NUL
MAPFILE_R = NUL
CC = cl
CFLAGS_G = /W3 /G2 /Zp /BATCH /FR$*.sbr
CFLAGS_D = /qc /Gs /Gi$(PROJ).mdt /Zr /Zi /Od
CFLAGS_R = /Ot /Oi /O1 /Oe /Og /Gs
ASM = masm
AFLAGS_G = /Mx /T
AFLAGS_D = /Zi
BRFLAGS = /o $(PROJ).bsc
BROWSE = 1
CVFLAGS = /50

OBJJS = MACHINE.obj
SBRS = MACHINE.sbr

all: $(PROJ).exe

.SUFFIXES: .c .sbr .obj

MACHINE.obj : MACHINE.C C:\C600\INCLUDE\os2.h C:\C600\INCLUDE\stddef.h\
machine.h C:\C600\INCLUDE\os2def.h C:\C600\INCLUDE\bse.h\
C:\C600\INCLUDE\pm.h C:\C600\INCLUDE\bsedos.h C:\C600\INCLUDE\bsesub.h\
C:\C600\INCLUDE\bseerr.h C:\C600\INCLUDE\bsedev.h\
C:\C600\INCLUDE\pmwin.h C:\C600\INCLUDE\pmgpi.h\
C:\C600\INCLUDE\pmdev.h C:\C600\INCLUDE\pmavio.h\
C:\C600\INCLUDE\pmpic.h C:\C600\INCLUDE\pmord.h\
C:\C600\INCLUDE\pmbitmap.h C:\C600\INCLUDE\pmfont.h\
C:\C600\INCLUDE\pmtypes.h C:\C600\INCLUDE\pmm1e.h\
C:\C600\INCLUDE\pmsh1.h C:\C600\INCLUDE\pmerr.h\
C:\C600\INCLUDE\pmhelp.h

MACHINE.sbr : MACHINE.C C:\C600\INCLUDE\os2.h C:\C600\INCLUDE\stddef.h\
machine.h C:\C600\INCLUDE\os2def.h C:\C600\INCLUDE\bse.h\
C:\C600\INCLUDE\pm.h C:\C600\INCLUDE\bsedos.h C:\C600\INCLUDE\bsesub.h\
C:\C600\INCLUDE\bseerr.h C:\C600\INCLUDE\bsedev.h\
C:\C600\INCLUDE\pmwin.h C:\C600\INCLUDE\pmgpi.h\
C:\C600\INCLUDE\pmdev.h C:\C600\INCLUDE\pmavio.h\
C:\C600\INCLUDE\pmpic.h C:\C600\INCLUDE\pmord.h\

```

```

C:\C600\INCLUDE\pmbitmap.h C:\C600\INCLUDE\pmfont.h\
C:\C600\INCLUDE\pmtypes.h C:\C600\INCLUDE\pmmle.h\
C:\C600\INCLUDE\pmsh1.h C:\C600\INCLUDE\pmerr.h\
C:\C600\INCLUDE\pmhelp.h

```

```

$(PROJ).bsc : $(SBRS)
    $(PWBRMAKE) @<<
$(BRFLAGS) $(SBRS)
<<

$(PROJ).exe : $(OBS)
!IF $(DEBUG)
    $(LRF) @<<$(PROJ).1rf
$(RT_OBS: = +^
) $(OBS: = +^
)
$@
$(MAPFILE_D)
$(LLIBS_G: = +^
) +
$(LLIBS_D: = +^
) +
$(LIBS: = +^
)
$(DEF_FILE) $(LFLAGS_G) $(LFLAGS_D);
<<
!ELSE
    $(LRF) @<<$(PROJ).1rf
$(RT_OBS: = +^
) $(OBS: = +^
)
$@
$(MAPFILE_R)
$(LLIBS_G: = +^
) +
$(LLIBS_R: = +^
) +
$(LIBS: = +^
)
$(DEF_FILE) $(LFLAGS_G) $(LFLAGS_R);
<<
!ENDIF
!IF $(DEBUG)
    $(ILINK) -a -e "$(LINKER) @$(PROJ).1rf" @$@
!ELSE
    $(LINKER) @$(PROJ).1rf
!ENDIF
$(NMAKEBSC1) MAKEFLAGS=
$(NMAKEBSC2) $(NMFLAGS) -f $(PROJFILE) $(PROJ).bsc

```

```

.c.sbr :
!IF $(DEBUG)
    $(CC) /Zs $(CFLAGS_G) $(CFLAGS_D) /FR$@ $<
!ELSE
    $(CC) /Zs $(CFLAGS_G) $(CFLAGS_R) /FR$@ $<
!ENDIF

.c.obj :
!IF $(DEBUG)
    $(CC) /c $(CFLAGS_G) $(CFLAGS_D) /Fo$@ $<
!ELSE
    $(CC) /c $(CFLAGS_G) $(CFLAGS_R) /Fo$@ $<
!ENDIF

run: $(PROJ).exe
    $(PROJ).exe $(RUNFLAGS)

debug: $(PROJ).exe
    CVP $(CVFLAGS) $(PROJ).exe $(RUNFLAGS)

```

**The Project File.** Very often an application is the sum of one executable module and one or more DLLs; each module being based on a single make file. The whole collection of make files can be easily managed in a project file, which is a container of one or more make files. In IBM WorkFrame/2 the distinction leads to base or composite project files.

**The Module Definition File.** A PM application requires a file with the DEF extension. This file tells the linker how to behave during the construction of the executable. The linker used to create PM executables is also able to generate character-based programs as well as DLLs.

```

Operating System/2 Linear Executable Linker
Version 2.01.005 Mar 16 1993
Copyright (C) IBM Corporation 1988-1993.
Copyright (C) Microsoft Corp. 1988-1993.
All rights reserved.

```

The information displayed by the execution of LINK386.EXE varies according to the version actually installed in the system. The syntax, however, is always the same and takes five arguments on the command line.

<i>Argument</i>	<i>Description</i>
Object modules	List of object modules involved in the construction of the executable
Run file	Name that will be given to the executable (usually corresponds to the first object module listed in the first argument)
List file	Name given to a file containing a map of all sources



Libraries	List of object libraries and import libraries needed by the linker
Definition file	Module definition file

The first four items are well-known by any DOS programmer. What is specified in the DEF file is actually the result of the general organization of the application in separate source code modules (i.e. separate files with the C extension), with respect to the most probable frequency of usage. The module definition file can be written with a plain text editor, and lists a series of directives that are meaningful to the linker. These directives are summarized in Table 2.1. In Listing 2.3 you can see a simple module definition file.

### Listing 2.3 A Generic Module Definition File for the OS/2 PM Linker

---

```

;-----
; MACHINE.DEF module definition file
;-----

NAME      MACHINE
DESCRIPTION 'Two windows and two classes'
PROTMODE
HEAPSIZE  1024
STACKSIZE 8192

CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MULTIPLE

EXPORTS
  ClientWndProc1
  ClientWndProc2

```

The value specified by the `HEAPSIZE` directives sets the overall size of the program's *local heap* inside the *private data area*. In OS/2 1.3, this same directive referred to the initial local heap size, with a theoretical expansion limit of up to 64KB. You must be very careful when porting old 16-bit OS/2 code, because an undersized heap might bring unexpected results.

It is important to remember that when writing a module definition file you must always list with an `EXPORTS` directive the names of all functions declared, such as `EXPENTRY` in the application code. However, if you are producing an EXE, this step is unnecessary. In the module definition files in the rest of this book you will find all `EXPENTRY` functions declared in an `EXPORTS` directive. This is simply good programming practice. Chapter 9 covers the DEF file in great detail.

## Resource Files

To create the resource file needed in the development of a PM application requires a specific tool, the resource compiler `RC.EXE`. The term *resource* refers to modular

**Table 2.1 The Directives in a Module Definition File**

<i>Directive</i>	<i>Description</i>
BASE	Generates the addresses of all items in a module on multiples of 64KB.
CODE	Attributes for the application's code segments: PRELOAD   LOADONCALL EXECUTEONLY   EXECUTEREAD IOPL   NOIOPL CONFORMING   NONCONFORMING
DATA	Attributes for the application's data segments: PRELOAD   LOADONCALL READONLY   READWRITE NONE   SINGLE   MULTIPLE IOPL   NOIOPL SHARED   NONSHARED
DESCRIPTION	Text description enclosed in single quotes.
EXETYPE	Identifies the kind of executable: OS2   WINDOWS   UNKNOWN
EXPORTS	Lists the exported functions that have to be executed with IOPL.
IMPORTS	Lists the imported functions coming from other modules, usually DLLs.
HEAPSIZE	Size, in bytes, of the application's local heap.
LIBRARY	Name of the module that will become a DLL.
NAME	Name of an executable.
PHYSICAL DEVICE	Identifies a physical device driver.
PROTMODE	Defines the module as a protected mode executable.
SEGMENTS	Defines the segment attributes of the segments that make up the application.
STACKSIZE	Size, in bytes, of the application's stack.
STUB	Adds a DOS stub module to an OS/2 executable.
VIRTUAL DEVICE	Identifies a virtual device driver.

portions of the applications that can be described by means of a textual representation (*template*) or other binary files generated with specific utilities.

A resource file—which has the extension RC—is a plain ASCII text file containing two kinds of resources: textual and binary. The first kind of resource includes, among others, the descriptions (*template*) of menus, dialog boxes (*dialog template*), and

windows (*windows template*), blocks of strings of text (*string table*) or messages (*message table*) displayed during the execution of the application. Binary resources, on the other hand, include bitmaps, icons, cursors, and fonts. Listing 2.4 is an example of a generic resource file.

#### Listing 2.4 A Typical Resource File Containing a Menu Template and a Dialog Template

---

```
// TWOMENUS.RC

#define INCL_WIN
#include <os2.h>
#include "twomenus.h"

ICON RS_ICON TWOMENUS.ICO

MENU RS_MENU
{
    SUBMENU "~File" , MN_FILE
    {
        MENUITEM "~New", MN_NEW
        MENUITEM "~Open", MN_OPEN
        MENUITEM "~Save", MN_SAVE
        MENUITEM "Save ~As...", MN_SAVEAS
        MENUITEM SEPARATOR
        MENUITEM "E~xit", MN_EXIT
    }
    ...
}
...

STRINGTABLE
{
    ST_CLASSNAME, "TWOMENUS"
    ST_WINDOWTITLE, "Loading a second menu"
}
...
```

The resource file is a novelty if you are used to programming in DOS or in OS/2 kernel, but it might also be an old friend if you have previously programmed in MS Windows or in OS/2 1.x. There are two reasons that a resource file is needed in a PM application. Because PM is a multitasking environment, it is possible that RAM might get overloaded with different code and data. The Intel 32-bit processors can manage up to 4GB of physical memory, but this is only a theoretical quantity. It is far more likely that an OS/2 machine will be equipped with much less memory, usually between 6 and 8MB of RAM. Minimizing memory swapping is important for optimal overall performance. Applications exploiting resource files may load some module portions into memory at different times than when they are actually started.

Furthermore, resource files also meet the growing need for international versions of applications. The translation process may be limited to rewriting in the various target languages all text items that appear in a source file, preserving the basic directives of the original version as far as memory requirements and program logic are concerned. By examining Listing 2.4 and Figure 2.4 which shows the flow chart of the development of a PM application, you can see that the translation process is easy and safe. No part of the source code is ever modified. Without changing the basic framework of an application, you can integrate new elements simply by acting upon the program's resources. For instance, you might define new icons, images, bitmaps, and other resources. Chapters 5, 6, and 8 will discuss the various kinds of resources that can be described in a RC file.

---

## Header Files

In addition to the header files provided by the development kit and the C compiler, every PM application has one or more header files containing defines, macros, and new data types. A resource file contains several items defined by the programmer, like those printed in *italic* in the following code fragment extracted from Listing 2.4:

```
// MENUAPP.RC

#define INCL_WIN
#include <os2.h>
#include "menuapp.h"

ICON RS_ICON MENUAPP.ICO

...
STRINGTABLE
{
    ST_CLASSNAME, "MENUAPP"
    ST_WINDOWTITLE, "A window with a menubar"
}
...
```

The strings in *italic* are the result of the usage of a preprocessor `#define` directive. These strings are used both in the resource file and in the source code. The usefulness of a header file containing all the descriptions of these text strings is twofold, because the header file can be included both in the RC file and in the C file. In this way you can save time by avoiding retyping information and be certain that you have consistent information in both files. Listing 2.5 reports the defines pertaining to the resource file examined earlier.

---

### Listing 2.5 A Portion of a Typical Header File

```
// MENUAPP.H

#define RS_ALL          300
```

```

#define RS_ICON          RS_ALL
#define RS_MENU          RS_ALL
#define RS_ACCELTABLE   RS_ALL
#define RS_TBMENU        350
...
#define ST_CLASSNAME     10
#define ST_WINDOWTITLE   11
...

```

---

## The Source Code

Source code plays the main role in a PM application. Like any other program written in C, it is possible to have multiple source files that generate the same number of object modules. The OBJ files are later linked together in a single executable application. For the moment we will assume a single source code file.

The development model of a PM application is somewhat complex and requires some degree of expertise before you can master all its elements. The following list summarizes the various essential phases needed for generating a PM application from scratch:

1. Include the OS2.H file at the beginning of the source file, following the correct INCL\_ directive listed in Appendix A.
2. Write all function prototypes used in the application.
3. Write all the definitions and macros useful in the source code.
4. Initialize the application.
5. Create the application's message queue.
6. Register one or more window classes specific to the program.
7. Create the main window which will identify the application once it is displayed on the screen.
8. Write the message loop.
9. Write as many window procedures as there are class registrations.
10. Pass all messages received in a window procedure to a function that provides their default processing.

When writing the source code, the first operation is to state which header files are needed by the program. It is good programming practice to precede the indication of OS2.H with one or more of the #define directives in Appendix A to specify which portions of the header file should be included.

```

#define INCL_WIN
#include <os2.h>
...

```

Appendix A lists all the INCL\_ defines contained in the OS/2 2.1 Toolkit header file.

Before moving on to the actual writing of the functions that make up the code, it is necessary to declare all identifiers with *file scope* (those that are visible from the point of declaration until the end of the file), and denote explicitly the prototypes of all functions to be found in the application. Function prototyping must be explicit in order to avoid compilation errors that would hinder the whole construction process.

A function prototype is the complete header of a function found in the application. The prototype also specifies the identifiers that will be used in the program.

```
#define INCL_WIN
#include <os2.h>

// function prototypes
int main( void) ;
MRESULT EXPENTRY ClientWndProc( HWND, ULONG, MPARAM, MPARAM) ;
...
```

The identifiers appearing in a function prototype assume function prototype scope:

```
...
MRESULT EXPENTRY ClientWndProc(   HWND hwnd,
                                   ULONG msg,
                                   MPARAM mp1,
                                   MPARAM mp2) ;
...
```

This second kind of layout will make the source code's functions' syntax even more readable. The return value of *ClientWndProc()*, as well as each of the four parameters, corresponds to one of the data types examined in Chapter 1.

---

## The *Main()* Function

The execution entry point for any PM application is represented by the *main()* function, as one can see from the declarations of the functions' prototypes. Just like any program written in C, *main()* requires parameters to catch the command line arguments and to reference the environment variables defined in the system's configuration files or in other batch files. In this case, the signature of the *main()* function looks like the following code fragment:

```
int main( int argc, char *argv[], char *envp[])
{
    ...
}
```

You might also use the *DosGetInfoBlocks()* function, which is part of the OS/2 Control Program API, in order to access the command line arguments as well as the

global environment variables. To use this function, it is necessary to issue an `INCL_DOS` directive or, more precisely, `INCL_DOSPROCESS` so that the appropriate function prototype is readily available and avoids compilation errors. The simplest PM program can thus be reduced to the following listing:

```
int main( void)
{
}
```

The effects of this listing are obviously very limited. So is it really a PM application? No, because a PM application must at least show a window on the screen with which the user can interact. A more precise definition of a PM application would require the presence of a *message queue*, and, as a consequence of such a queue, a possible *context switch* from a character screen group to that of PM.

It is therefore necessary to put some PM API calls inside the body of the *main()* function in order to create the message queue. These API calls are all characterized by the *Win* prefix. However, before doing this it is necessary to have an *anchor block*. The meaning of an anchor block is limited in PM. It is simply the return value of the *WinInitialize()* function that has to be called as the very first function in a PM thread. The handle of an anchor block is required as a parameter in many other PM API calls, but it can just as well be replaced by the `NULLHANDLE` definition. In this case it is not really a piece of information associated with a process, but simply an artifact of the SAA rules for code and application portability between different platforms.

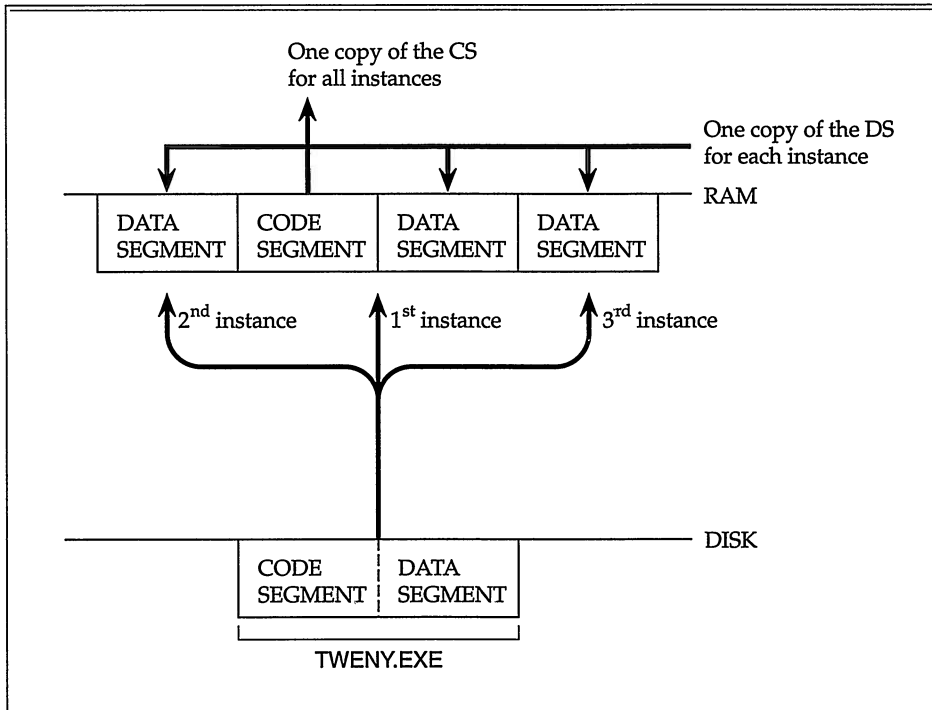
PM processes, just like any other OS/2 task, are identified by a unique PID and a TID for every thread. More precisely, the term *process* for a PM executable should be replaced by the term *instance*, which indicates more accurately the uniqueness of execution of an application. This distinction is essential in PM because the EXE is actually executable in more than one copy simultaneously, all of which are active on the screen at the same time. Figure 2.6 demonstrates the concept of instances, the associated use of a unique PID to provide a distinct item to each instance, and the presence of one single code segment. Otherwise, two copies of the same program would be indistinguishable if you were to reference them by the same executable module.

## Initialization of an Instance

The *WinInitialize()* function returns a handle to an *anchor block*, provided you specify a parameter that always has the value zero:

```
#define INCL_WINWINDOWMGR
HAB APIENTRY WinInitialize( USHORT fsOptions) ;
```

<i>Parameter</i>	<i>Description</i>
fsOptions	Initialization option (always zero)
<i>Return Value</i>	<i>Description</i>
HAB	Handle to the application's anchor block



**Figure 2.6** The running of multiple instances of the same application in PM.

It is common practice to declare the identifier `hAb` having *block scope*, that is directly in the `main()` function, even though its usage spans more than one function in a PM application. This approach, which is followed in this book, allows you to avoid using file scope identifiers in the source code. If it were necessary to retrieve again the process's anchor block in any code fragment different than `main()`, it would be necessary to resort to the `WinQueryAnchorBlock()` function, which is designed specifically for this purpose:

```
#define INCL_WININPUT
HAB APIENTRY WinQueryAnchorBlock( HWND hwnd) ;
```

<i>Parameter</i>	<i>Description</i>
<code>hwnd</code>	Handle of a window created in the application

<i>Return Value</i>	<i>Description</i>
<code>HAB</code>	Handle to the application's anchor block

Here `hwnd` is the handle of a generic window of the application (as we will see in Chapter 4, any window can be specified as the parameter to this function).



## Creating the Message Queue

Once you have the anchor block handle, you proceed by creating the application's message queue with the `WinCreateMsgQueue()` function. Since this action requires you to have `hab` readily available, and that the value of this identifier be specific for the instance at hand, the creation of the message queue is an instance-specific task for every PM application. (The one significant exception being multithreaded code, which is legitimate and even recommended in PM. This approach will be examined in Chapter 9.) The syntax of `WinCreateMsgQueue()` is as follows:

```
#define INCL_WINMESSAGEMGR
HMQ APIENTRY WinCreateMsgQueue( HAB hab, LONG msg);
```

<i>Parameter</i>	<i>Description</i>
<code>hab</code>	Handle to the application's anchor block
<code>msg</code>	Application's message queue size: zero means 10 messages

<i>Return Value</i>	<i>Description</i>
HMQ	Handle to the application's message queue

The first parameter is the anchor block handle; the second one is a `LONG` value which indicates the size of the queue in terms of number of messages it will be able to accommodate. If you specify zero, then the default value of 10 messages will be used. The return value of this function is a handle to the message queue; the number of message queues is equal to one for every thread in PM.

Before expanding on the subject of messages, a fundamental element in the working of PM (that will be discussed in Chapter 3), it is necessary to stress another feature of `WinCreateMsgQueue()`. This function forces a context switch from a character based screen group to that of PM every time it might be necessary. In practical terms, this qualifies the executable module to the system like a PM application.

## Registering a Window Class

PM is an environment featuring multiple windows with different sizes and appearances defined by the combination of specific styles and attributes. Each window created by a program belongs to a *window class*, which can be thought of as a category of generic window that must be described by the programmer.

To illustrate this concept, let's look away from computers for a moment. The term *flower* identifies several thousand different species of flowers, all of which share some common morphologic and phenotypical characteristics. Each and every single flower is a unique materialization of the general idea of a flower.

This same concept exists in PM; the system must be able to deal with the "idea" of some kind of window in order to be able to create a real window that will be displayed on the screen.

PM has fifteen predefined windows classes—categories of windows that are already registered and thus immediately available by means of specific API functions of PM.

This means that PM possesses different “ideas” of windows, each corresponding to some kind of window that the programmer creates in the code without having to set them up through the preliminary class registration.

Individual programmers will at some time wish to define new kinds of windows better suited to their program’s specific requirements. The windows belonging to the predefined window classes are very useful whenever control windows or dialog windows (Chapter 8) are involved, and are very seldom used to implement and display entire applications on the screen.

In order to register a new window class, use the *WinRegisterClass()* function:

```
#define INCL_WINWINDOWMGR
BOOL APIENTRY WinRegisterClass(  HAB hab,
                                PSZ pszClassName,
                                PFNWP pfnWndProc,
                                ULONG flStyle,
                                ULONG cbWindowData);
```

<i>Parameter</i>	<i>Description</i>
hab	Handle to the application’s anchor block, previously returned by <i>WinInitialize()</i> .
pszClassName	Name of the new window class that needs to be registered. This may be any name you please, as long as it is unique within each instance.
pfnWndProc	Name of the function that will act as the window procedure for the class being registered.
flStyle	Set of styles (defines introduced by the CS_ prefix) that qualify the behavior and the appearance of the windows belonging to this class.
cbWindowData	Number of bytes available to a buffer reserved for each window belonging to the class being registered.
<i>Return Value</i>	<i>Description</i>
BOOL	Success/failure of the operation.

There is no limit to the name you can assign to a window class, with the exception of those names already used for the predefined windows of PM. A common practice is to assign to the class of the application’s main window the same name as the executable module. Thus, if you are writing the application TWENY, you will most likely have a window class with the same name.

By definition, classes in PM are private—visible and usable exclusively from within an application instance, and not from within any other executable module. Any attempt to register an identical window class within one same instance produces a null return value, thus aborting any further attempt.

The third parameter of *WinRegisterClass()* is a pointer to a function, as defined in PMWIN.H and shown in Table 1.9 of Chapter 1. This identifier is of type PFNWP and the function to which it points returns a value whose type is MRESULT. This function

takes four parameters of type HWND, ULONG, MPARAM, and MPARAM. The signature of a generic window procedure is the following one:

```
MRESULT EXPENTRY xxx(    HWND hwnd,
                        ULONG msg,
                        MPARAM mp1,
                        MPARAM mp2)
{
    ...
}
```

where xxx is the name chosen by the programmer. Even in this case, there are no naming rules, and imagination is the only real limitation. In this book, however, we will always label the window procedure of the window class to which the main application window belongs to with a name like *ClientWndProc()*.

The purpose of the pointer in *WinRegisterClass()* is to provide the system (PM) the address of the window procedure that will receive all messages issued by the interactions made by a user with a window belonging to that specific window class. This delivers a high degree of flexibility to the system, as you will better understand once we get to the chapter on windowing (Chapter 4).

The element that needs to be stressed is the remarkable correlation that exists between a window class and its own window procedure. All messages generated by interacting with any window belonging to that class will invariably be forwarded by the system to the function indicated at the time of registration.

Furthermore, each class has individual attributes that affect the behavior of every single window created by the program and belonging to that class. The last two parameters, in fact, allow you to specify styles that are unique to this window class, as well as any additional memory required by each window.

## Class Styles

The styles of a window class are represented by bits in a ULONG value. Therefore, there are 32 possible values that can be combined in various ways. Table 2.2 lists and describes the effects of each style currently available. The numeric values in the table correspond to what is actually defined in PMWIN.H.

Styles primarily govern the behavioral aspects of windows, rather than their appearance. The setting of a style during the registration phase of a class means that all windows belonging to that class will automatically possess the corresponding attribute.

## Window Words

The final parameter, *cbWindowData*, defines a number of additional bytes that will augment the size of a memory buffer available to each window. This aspect of the development model in PM is examined in Chapters 4 and 11, when we explore how to develop MDI (Multiple Document Interface) applications. When a window is created, PM automatically allocates it a memory block. This area contains information

**Table 2.2 The CS\_ Styles That Can Be Specified During a Window Class Registration**

<i>Style</i>	<i>Value</i>	<i>Description</i>
CS_CLIPCHILDREN	0x20000000L	The windows belonging to a class that has this style flag set will automatically have the WS_CLIPCHILDREN style at the moment when they are created.
CS_CLIPSIBLINGS	0x10000000L	The windows belonging to a class that has this style flag set will automatically have the WS_CLIPSIBLINGS style at the moment when they are created.
CS_FRAME	0x00000020L	The windows belonging to a class that has this style flag set will receive the message WM_FRAMEDESTROY just before being destroyed.
CS_HITTEST	0x00000008L	Will force the system to send the WM_HITTEST message every time the mouse is moved.
CS_MOVENOTIFY	0x00000001L	Will force the system to send the WM_MOVE message every time the window is moved.
CS_PARENTCLIP	0x08000000L	The windows belonging to a class that has this style flag set will automatically have the WS_PARENTCLIP style at the moment when they are created.
CS_PUBLIC	0x00000010L	Creates a public window class, i.e. a window class that is accessible and usable from within any PM application.
CS_SAVEBITS	0x04000000L	The windows belonging to a class that has this style flag set will automatically have the WS_SAVEBITS at the moment when they are created.
CS_SIZEREDRAW	0x00000004L	Will force the system to invalidate the entire client window every time the size of the window is changed.
CS_SYNCPAINT	0x02000000L	The windows belonging to a class that has this style flag set will automatically have the WS_SYNCPAINT style at the moment when they are created.

that qualifies the window in terms of its position on the screen as well as other properties. During the registration phase, the designer can define and extend this area. The purpose is that of getting a place to store application-specific pieces of data.

```

...
if( !WinRegisterClass( hab,
                      szClassName,
                      ClientWndProc,
                      CS_SIZEREDRAW,
                      0L) ;

    return FALSE ;

...

```

---

## The Predefined Window Classes

PM has as many as fifteen predefined window classes. These classes are able to generate extremely specialized windows that are very useful when dealing with specific application needs. For example, the class `WC_SCROLLBAR` allows you to create optimized graphics tools that allow you to command horizontal or vertical scroll operations on the contents of a window. The class `WC_ENTRYFIELD` controls text string input and output. A window of this class is handy when you need to prompt the user for his name and password in order to log on to a network server. A detailed analysis of the predefined window classes appears in Chapter 7. Table 2.3 lists all the preregistered window classes available in PM.

**Table 2.3 List of Preregistered Window Classes Available in PM**

<i>Class</i>	<i>Value</i>	<i>Description</i>
<code>WC_BUTTON</code>	<code>((PSZ)0xffff0003L)</code>	Any kind of button displayed in a PM application belongs to this class: push buttons, radio buttons, check boxes, and user defined buttons.
<code>WC_COMBOBOX</code>	<code>((PSZ)0xffff0002L)</code>	The windows of this class are a kind of combination between an entry field and a list box, and are a convenient means for keyboard input with very limited on-screen space requirements.
<code>WC_CONTAINER</code>	<code>((PSZ)0xffff0025L)</code>	Window to implement folders in WPS.
<code>WC_ENTRYFIELD</code>	<code>((PSZ)0xffff0006L)</code>	The windows of this class are specifically designed to deal with user input of alphanumeric text strings.
<code>WC_FRAME</code>	<code>((PSZ)0xffff0001L)</code>	This class covers any kind of frame window.

*(continued)*

**Table 2.3 (Continued)**

<i>Class</i>	<i>Value</i>	<i>Description</i>
WC_LISTBOX	((PSZ)0xffff0007L)	The windows of this class are specifically designed for displaying a number of text or bitmap items, albeit in a limited on-screen space.
WC_MENU	((PSZ)0xffff0004L)	All PM menus belong to this class. The availability of a specific class for menu allows to insert into the client window other windows that are structured just like menus.
WC_MLE	((PSZ)0xffff000aL)	This class encompasses all windows that deal with input activities. These windows are similar to entry fields, with the difference that they can span several lines of text.
WC_NOTEBOOK	((PSZ)0xffff0028L)	These windows take on the appearance of a notebook, with several pages.
WC_SCROLLBAR	((PSZ)0xffff0008L)	These windows are the scroll bars that help manage the client window, in practice by extending its size beyond those physically available on the screen.
WC_SLIDER	((PSZ)0xffff0026L)	These windows can take on different looks and can represent visually the progress of time, of task completion or other kind of measurements.
WC_SPINBUTTON	((PSZ)0xffff0020L)	This kind of window is similar to an entry field, with the addition of two arrows, pointing up and down, which can be used to control the value in the field.
WC_STATIC	((PSZ)0xffff0005L)	This kind of window is used to display static text; it's usage is very common inside dialog windows (Chapter 8).
WC_TITLEBAR	((PSZ)0xffff0009L)	Class of windows used mainly as structural elements to allow the user to move windows and to display their titles.
WC_VALUESET	((PSZ)0xffff0027L)	Windows made of different panes containing text strings, bitmaps, icons, colors, or other items defined by the programmer.

## Why Register Window Classes?

The availability of predefined window classes in PM might make you wonder why you need to register any additional classes. Note that all preregistered classes refer to some specific kind of window, like *comboboxes* or *listboxes*, which are useful in writing an application's output routines, because they are designed for that kind of task. The software designer, however, may need to have more generic windows to fully meet the requirements of the program. The registration of a window class, that is, passing to PM information on a new category of possible windows, meets this need. In essence, registering a class means transferring pieces of information to PM: these data are exactly the parameters of *WinRegisterClass()*. The storage of these data does not take place in the data segment of the application that calls *WinRegisterClass()*. Instead, it occurs directly in the data segment of PM.

## The Nature of PM

The graphical user interface of OS/2 systems is implemented by means of a number of DLLs present in the \OS2\DLL directory of the boot disk. There are more than twenty, but the exact number varies according to the system's configuration (version 2.0 GA, installation of the service pack, or version 2.1). Among these, you will find PMWIN.DLL, which is the *dynamic link library* containing all functions that have the *Win* prefix and that establish the API of the windowing interface. Functions like *WinCreateStdWindow()* and *WinRegisterClass()* are to be found in this module. Other parts of what one might refer to in generic terms as PM are to be found in different DLLs. As far as the predefined window classes are concerned, for instance, some of their window procedures are placed in PMCTLS.DLL (Table 2.4).

All DLLs listed in Table 2.4 are always loaded directly into the system when it is booted; thus all fifteen preregistered window classes are immediately available. There are other specific initialization functions that do not appear in Table 2.4, and that issue calls to *WinRegisterClass()*. Similar to predefined window classes, those registered from within applications get stored inside PM, rather than in the data segment of the

**Table 2.4 List of Some Window Classes and the Corresponding Names of Their Window Procedures and the Modules in Which They Are Placed**

<i>Class</i>	<i>Window Procedure</i>	<i>Module</i>
WC_BUTTON	ButtonProc	BUTTON.DLL
WC_MLE	MULTILINEEDITPROC	PMMLE.DLL
WC_NOTEBOOK	BOOKWNDPROC	PMCTLS.DLL
WC_SLIDER	SLIDERWNDPROC	PMCTLS.DLL
WC_SPINBUTTON	SPINBUTTONWNDPROC	PMCTLS.DLL
WC_VALUESET	VALUESETWNDPROC	PMCTLS.DLL

process that issued the registration. This solution, as we will see in Chapter 3, is the foundation of the windowing mechanism of PM.

## Strategies for Registering Window Classes

In a generic application, you can register several classes, each one of which is specialized in delivering a specific kind of window. For instance, let's examine Microsoft Excel 3.0 for OS/2. This application is easily recognized by a top-level window with the title MS Excel that belongs to the class XLMAIN registered by the application. All worksheets are displayed in a different kind of window (EXCEL4), graph windows (EXCEL6), and windows for writing macros (EXCEL5). The reference cell has a window class of its own (EXCEL1), and so has the window for setting commands (EXCEL2), the status bar (EXCEL3) and the client window of the application (XLDESK).

A class will characterize a number of windows that usually have a window procedure in common. An application usually registers several window classes. The whole process is outlined in Figure 2.7.

## Creating a Window

Once you have a window class specifically designed for the application you are developing, you create the window so as to verify the application-window identity. As you will see later, there are various ways to create a window in PM. The easiest method is through the use of *WinCreateStdWindow()*. As the name implies, it creates a standard window. The function that should be examined first is *WinCreateWindow()*,

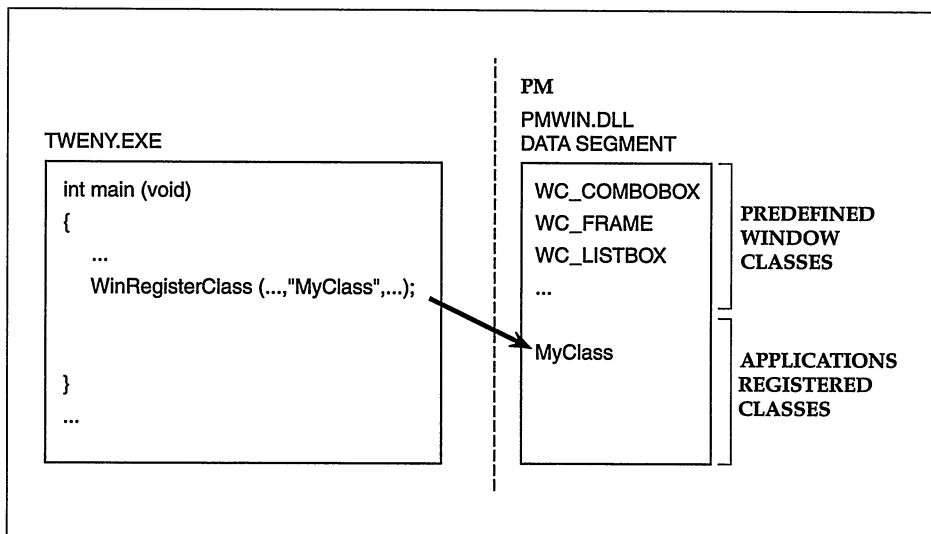


Figure 2.7 Registration of window classes in PM.



which is an even simpler tool for creating a window. This method, however, requires you to understand some concepts that will be discussed later in the book. Therefore, we now focus on *WinCreateStdWindow()*:

```
#define INCL_WINFRAMEMGR
HWND APIENTRY WinCreateStdWindow(  HWND hwndParent,
                                   ULONG fStyle,
                                   PULONG pflCreateFlags
                                   PSZ pszClientClass,
                                   PSZ pszTitle,
                                   ULONG styleClient,
                                   HMODULE hmod,
                                   ULONG idResources,
                                   PHWND phwndClient) ;
```

<i>Parameter</i>	<i>Description</i>
hwndParent	Handle of the parent window
fStyle	Styles of the window's frame window
pflCreateFlags	Parameters passed to the window's frame window
pszClientClass	Name of the class to which the window belongs
pszTitle	Window title (if the titlebar is present)
styleClient	Style of the client window
hmod	Handle to the resource management module
idResources	Resource ID to be associated to the window
phwndClient	Handle of the client window
<i>Return Value</i>	<i>Description</i>
HWND	Frame window handle or NULLHANDLE in case of error

This function creates a window and returns a handle that will allow references to it in various circumstances. (Actually, the true action performed by this function is to create multiple windows by calling *WinCreateWindow()* many times. For the moment, we can think of a window as a single object that you can manage simply by referring to its handle.) By varying the combination of values of the single parameters, you create windows that are very different from one another in appearance as well as in function. We will return to this subject in Chapter 4.

## *The Parameters of WinCreateStdWindow()*

Every PM window must have a parent; this explains why the first parameter is there. In this case we are trying to build a very simple PM application, and thus the only window that need be displayed on the screen can be thought of as being identical to the actual program. The parent of this window is the screen background, just like any other top-level window. The value to assign to the first parameter is therefore `HWND_DESKTOP`, which corresponds to the PM screen group's background, as defined in `PMWIN.H`.

Once the window's parent has been indicated, it is necessary to specify its style by determining the attributes that affect both its look as well as its function. The `ULONG`

type parameter allows you to define the various styles by selectively setting the bits in this 32-bit value. You can identify the various flags in this parameter by means of identifiers that have a WS\_ (*window style*) or FS\_ (*frame style*) prefix, as indicated in Tables 2.5 and 2.6.

**Table 2.5 Window Styles (WS\_) Acceptable by the *WinCreateStdWindow()* Function**

<i>Style</i>	<i>Value</i>	<i>Description</i>
WS_ANIMATE	0x00400000L	Enables animation effects when the window is created.
WS_MAXIMIZED	0x00800000L	Assigns the whole screen area to the window (the window is maximized).
WS_MINIMIZED	0x01000000L	Reduces the window to its own icon (the window is minimized).
WS_SYNCPAINT	0x02000000L	Alters the default behavior of a generic window as far as the handling of the WM_PAINT message is concerned. This message notifies the need to redraw some portion of the window's client area. By specifying this style, a WM_PAINT message in the message queue will immediately invalidate a portion of the client window.
WS_SAVEBITS	0x04000000L	Before the window is displayed on the screen, this flag will store the image corresponding to the pixels that will be occupied by the new window. When the new window is moved or closed, the uncovered pixels will be automatically restored to their previous contents. Menu windows are a typical example of this.
WS_PARENTCLIP	0x08000000L	Prevents any child window from painting over the pixels that are exclusively owned by the parent window.
WS_CLIPSIBLINGS	0x10000000L	Prevents a window from painting any of its pixels that are currently covered by a window at the same hierarchical level in the system.
WS_CLIPCHILDREN	0x20000000L	Prevents a window from painting any of its pixels that are currently covered by any of its child windows.
WS_DISABLED	0x40000000L	Disables the window so that it will no longer react to any mouse or keyboard action.
WS_VISIBLE	0x80000000L	Forces the window to display simultaneously to its creation. PM's API also provides alternative solutions for solving the problem of displaying a window.

**Table 2.6 The (FS\_) Window Styles Acceptable by the *WinCreateStdWindow()* Function**

<i>Style</i>	<i>Value</i>	<i>Description</i>
FS_ICON	0x00000001L	An icon is assigned to the window. The icon will be displayed in case the window is minimized. The icon's ID is the same as that used for the menu and the accelerator table associated with the windows.
FS_ACCELTABLE	0x00000002L	Assigns an accelerator table, as described in the application's resource file, to the frame window.
FS_SHELLPOSITION	0x00000004L	The window's position and size are assigned automatically by the system.
FS_TASKLIST	0x00000008L	The window's name, taken from the titlebar, will be inserted into the Task List application.
FS_STANDARD	0x0000000FL	Set of the following style flags: FS_ICON FS_ACCELTABLE FS_SHELLPOSITION FS_TASKLIST
FS_NOBYTEALIGN	0x00000010L	The window is positioned on screen with no byte alignment; this solution grants a greater flexibility in the positioning of the window, albeit it will reduce the speed of screen refresh. Therefore it is advisable to avoid this flag whenever there is a chance that the application might be installed on system with slow video adapters.
FS_NOMOVEWITHOWNER	0x00000020L	Will create a window that will not move even if its owner is moved.
FS_SYSMODAL	0x00000040L	Creates a system modal window that will capture exclusively the input focus.
FS_DLGBORDER	0x00000080L	Assigns a double border to the window which is typical of dialog windows and which does not allow resizing of the window; message boxes will have this style set by default.
FS_BORDER	0x00000100L	Assigns a single border to the window, which does allow resizing of the window.
FS_SCREENALIGN	0x00000200L	Creates a window which is aligned with respect to the screen (8-bit boundary).

*(continued)*

Table 2.6 (Continued)

Style	Value	Description
FS_MOUSEALIGN	0x00000400L	The window is positioned on the screen according to the current screen position of the mouse pointer; this solution minimizes a user's access time to the structural items available in the window.
FS_SIZEBORDER	0x00000800L	The window has a resizing border.
FS_AUTOICON	0x00001000L	Optimizes the window's repainting speed when it is minimized by not sending the WM_PAINT message to the client window.

There are many possible combinations. The WS\_ flags occupy the high word of the value, and the FS\_ flags the low word. Some WS\_ flags are automatically set by a previous CS\_ specified at the class's registration. The main difference between the two solutions is to be found in the greater degree of flexibility given by the usage of a WS\_ rather than a CS\_ flag. With a class style flag you force each window belonging to that class to always have that particular style, and limit the possibilities of diversifying among windows and implementation designs. Thus, when a class is registered, the number of styles defined is usually limited, because it is preferable to act at the level of the creation of every single window. In general, you will usually specify the style WS\_VISIBLE to display the window as soon as it is created (in fact, creating a window doesn't necessarily mean displaying it).

One use of *WinCreateStdWindow()*, that is suggested by the name assigned to the returned handle, is that of creating a *frame window*—a window belonging to the WC\_FRAME class. The flags specified in the second parameter always affect the frame window (FS\_), but can nonetheless be combined with those of a generic window (WS\_). The two styles expressed in the second parameter of *WinCreateStdWindow()* are somewhat general. The (WS\_) styles are useful but induce somewhat feeble effects. Instead, the (FS\_) styles are much "stronger." The FS\_STANDARD style alone can force *WinCreateStdWindow()* to search for an icon, a menu table, and an accelerator table in the application's resource file and to automatically associate these resources to the window being created. Therefore, as a good programming practice, especially when you have little experience, it is advisable to use WS\_ only, and save the FS\_ for special situations.

The third parameter is a pointer to an ULONG value containing various flags that affect structural and functional features of the window. These flags are distinguished by the FCF\_ (*frame creation flags*) prefix, and very often denote the same effects as the FS\_ styles. Table 2.7 summarizes and describes all FCF\_ flags present in OS/2 2.1.

**Table 2.7 The FCF\_ Styles Acceptable by the WinCreateStdWindow( ) Function**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
FCF_TITLEBAR	0x00000001L	Creates a window belonging to the WC_TITLEBAR class.
FCF_SYSMENU	0x00000002L	Creates the window's titlebar menu.
FCF_MENU	0x00000004L	Implies the creation of a new menu window that appears in the main window, right under the titlebar.
FCF_SIZEBORDER	0x00000008L	Assigns a border to the window so that the user can resize it.
FCF_MINBUTTON	0x00000010L	Creates an icon containing a downward pointing arrow (minimize icon).
FCF_MAXBUTTON	0x00000020L	Creates an icon containing an upward pointing arrow (maximize icon).
FCF_MINMAX	0x00000030L	Creates a pair of icons, one with a downward pointing arrow and one with an upward pointing arrow; it is equivalent to FCF_MINBUTTON   FCF_MAXBUTTON.
FCF_VERTSCROLL	0x00000040L	Assigns a vertical scrollbar to the window.
FCF_HORZSCROLL	0x00000080L	Assigns a horizontal scrollbar to the window.
FCF_DLGBORDER	0x00000100L	Creates a window featuring a double-border that is typical of dialog windows and that does not allow resizing of the window.
FCF_BORDER	0x00000200L	Creates a window featuring a single border that cannot be resized; the border's actual size depends on the resolution of the video adapter at hand.
FCF_SHELLPOSITION	0x00000400L	Establishes that the system will be responsible for defining the initial size and position of the window on the screen.
FCF_TASKLIST	0x00000800L	Adds the current window to the list of active tasks by specifying its name as it appears in the titlebar.
FCF_NOBYTEALIGN	0x00001000L	The client window of the window being created will not be byte-aligned; this may reduce screen refresh speed.
FCF_NOMOVEWITHOWNER	0x00002000L	Creates a window that will not move even though its owner is moved.

*(continued)*

**Table 2.7 (Continued)**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
FCF_ICON	0x00004000L	Assigns to the window an icon stored in the application's resource file and recognized by the ID specified as the eighth parameter to the <i>WinCreateStdWindow()</i> function.
FCF_ACCELTABLE	0x00008000L	Assigns to the window an accelerator table (Chapter 6) stored in the application's resource file and identified by the ID specified as the eighth parameter to the <i>WinCreateStdWindow()</i> function.
FCF_SYSMODAL	0x00010000L	Creates a system modal window, that will require special handling as far as input focus is concerned.
FCF_SCREENALIGN	0x00020000L	Establishes that the alignment of a window will be calculated according to the screen's actual dimensions.
FCF_MOUSEALIGN	0x00040000L	Places the window on the screen according to the position of the mouse pointer, thus making it easy to perform any subsequent selection.
FCF_STANDARD	0x0008CC3FL	Shorthand notation to indicate all of the following styles: FCF_TITLEBAR FCF_SYSMENU FCF_MENU FCF_SIZEBORDER FCF_MINMAX FCF_ICON FCF_ACCELTABLE FCF_SHELLPOSITION FCF_TASKLIST FCF_PALETTE_NORMAL
FCF_HIDEBUTTON	0x01000000L	Substitutes a hiding icon for a minimized icon.
FCF_HIDEMAX	0x01000020L	Combines a hiding icon with a maximizing icon.
FCF_AUTOICON	0x40000000L	Optimizes the performance of repaint operations whenever a window is minimized, without sending a WM_PAINT message to the client window (an operation that would just waste time).

## Comparison between FS\_ and FCF\_ Styles

The result achieved by naming a style with a FS\_ prefix is often equivalent to what can be achieved through a FCF\_ style, with some notable exceptions, as documented in Table 2.8.

We have already seen that *WinCreateStdWindow()* is actually a macro function, that is, a piece of code that really calls many other API functions of PM. Among them, the most important is *WinCreateWindow()*, followed by *WinLoadPointer()*, *WinLoadMenu()*, *WinLoadAcceltable()* and others. The FCF\_ flags, which are applicable only within *WinCreateStdWindow()*, will manifest their effects both on the frame window as well

**Table 2.8 Comparison between FS\_ and FCF\_ Styles**

<i>Frame Creation Flag</i>	<i>Frame Style</i>
FCF_TITLEBAR	Absent
FCF_SYSMENU	Absent
FCF_MENU	Absent
FCF_SIZEBORDER	FS_SIZEBORDER
FCF_MINBUTTON	Absent
FCF_MAXBUTTON	Absent
FCF_MINMAX	Absent
FCF_VERTSCROLL	Absent
FCF_HORZSCROLL	Absent
FCF_DLGBORDER	FS_DLGBORDER
FCF_BORDER	FS_BORDER
FCF_SHELLPOSITION	FS_SHELLPOSITION
FCF_TASKLIST	FS_TASKLIST
FCF_NOBYTEALIGN	FS_NOBYTEALIGN
FCF_NOMOVEWITHOWNER	FS_NOMOVEWITHOWNER
FCF_ICON	FS_ICON
FCF_ACCELTABLE	FS_ACCELTABLE
FCF_SYSMODAL	FS_SYSMODAL
FCF_SCREENALIGN	FS_SCREENALIGN
FCF_MOUSEALIGN	FS_MOUSEALIGN
FCF_STANDARD	Absent
FCF_HIDEBUTTON	Absent
FCF_HIDEMAX	Absent
FCF_AUTOICON	FS_AUTOICON
Absent	FS_STANDARD

as on other components of the window. The FS\_ flags, which are applicable both in *WinCreateStdWindow()* as well as *WinCreateWindow()*, affect only the appearance and the behavior of the frame window.

The FS\_ flags are seldom used with *WinCreateStdWindow()* because you will find in the FCF\_ more complete alternatives. The choice of the FCF\_ flags over the FS\_ flags involves some practical considerations. While it is always possible to specify a 0L in place of WS\_ and FS\_ styles (the second parameter to *WinCreateStdWindow()*), it is instead necessary to provide the address of a ULONG value when using the FCF\_ styles. Therefore, as you have to pass a value to the function, it is convenient to specify all flags needed directly in a FCF\_. (Also, the FS\_ flags have no value for automatically inserting the application's name into the Window List.)

For the moment we will assign to this parameter the value of FCF\_STANDARD, although subtracting the three styles FCF\_MENU, FCF\_ICON, and FCF\_ACCELTABLE. This is somewhat different from what one might expect to see with the accumulation of different styles by using a bitwise or (|), and it is just a lazy programmer's way of getting the most done by writing the least amount of code. In fact, FCF\_STANDARD is made up of seven different flags preset for creating a generic window; the chosen solution allows you to specify only four styles, and avoids some excessive typing:

```
...
ULONG ulFCFrameFlag = FCF_STANDARD &
                    ~FCF_MENU &
                    ~FCF_ICON &
                    ~FCF_ACCELTABLE ;
...
```

The definition of FCF\_STANDARD also contains references to three generic resources present in the file with the same ID: menus, icons, and accelerator table. At the present stage, we will create a PM application file without an RC file; therefore, all references to program items that do not exist must be subtracted from FCF\_STANDARD. The presence of equivalent FCF\_ flags together with the absence of resources would cause a disaster with unpredictable consequences.

The fourth parameter, *pszClientClass*, is the name of the window class to which the new window belongs. It must be a text string that identifies one of the predefined PM classes (this, though, rarely happens) or one of the programmer-defined classes that has been registered at earlier stages.

```
...
CHAR szClassName[] = "TWENY" ;
...
```

The next pointer, to an ASCIIZ string, refers to the title that you wish to display in the titlebar. It is good practice to store the window's title in an identifier, rather than specifying it directly in the syntax of *WinCreateStdWindow()*.

```
...
CHAR szWindowTitle[] = "Hi, guys!" ;
...
```



The following `ULONG` value, the sixth parameter, is employed for defining the styles of the window's client window. Once again, you can use the flags introduced by the `WS_` prefix, as this is by all means a true window. In this case, though, it is unworkable to specify `FS_` style flags, because the client window is not a `WC_FRAME` class window. This parameter will gain even greater importance when you develop applications that have active child windows in their client window.

The seventh and eighth parameters, despite being different types, have the same purpose: defining which resources should be associated with the window. The choice might range from a menu, to an icon, and an accelerator table. In Chapters 5 and 6 we will discuss the resources and structure of menus.

In general, the `hmod` handle takes on the `NULLHANDLE (0L)` value. It indicates that the possible resources to be tied to the window being built are to be found in the current executable module. With `idResources` you can specify an ID that describes one or more resources of the three kinds indicated (in this case, the resource file will have to contain a menu template, an accelerator table, and an icon, all of which are indicated by the very same ID). For the moment, we will specify zero for this parameter, thus indicating that no resources are to be tied to the window during its creation phase.

The meaning of the seventh and eighth parameters of `WinCreateStdWindow()` is exclusively connected to the `FCF_MENU`, `FCF_ICON`, and `FCF_ACCELTABLE` flags. If one or more of these flags are indicated as the third parameter, then the following two conditions must be met:

- You must have a resource file that contains the resources equivalent to the `FCF_` flag indicated (menu, icon, or accelerator table);
- The next to last parameter has to be assigned a numeric value corresponding to the ID of the resource or of the resource(s) previously indicated.

The last value to provide the `WinCreateStdWindow()` function is the address of an identifier of type `HWND` that will be used later to manage the client window. To summarize, let's examine the following code fragment:

```
...
hwndFrame = WinCreateStdWindow(  HWND_DESKTOP,
                                WS_VISIBLE,
                                &ulFCFrameFlags,
                                szClassName,
                                szWindowTitle,
                                0L,
                                NULLHANDLE,
                                0L,
                                &hwndClient) ;
...
```

The desktop acts as the parent window, and the window is to be made visible. No resources are tied to this window.

## Some Remarks on *WinCreateStdWindow()*

*WinCreateStdWindow()* is an optimal solution for creating a window with several structural elements, like a menu, an icon, and an accelerator table. The developer must check that the handle returned by the function is valid (any positive number), because this value is the basis for most of the application's logic. Once a window has been created, it can be displayed on the screen, thus allowing the user to interact with it and perform the actions desired.

Among the nine parameters of the function, there is no item that sets the window's size and position on the screen. The `FCF_SHELLPOSITION` flag, included in `FCF_STANDARD`, is a workaround that solves the problem by giving the window a default position, width, and height.

## Errors with *WinCreateStdWindow()*

There are many potential application errors with the *WinCreateStdWindow()* function. One of the principal reasons for this is the use of an inexistent window class name or simply a misspelled predefined window class. The system cannot proceed and create a window, so nothing appears on the screen. This is why it is important to check the return value of *WinCreateStdWindow()* and manage any kind of error condition by terminating the program's execution whenever a zero handle is returned.

Furthermore, when you specify any `FCF_STANDARD` flag, it is also necessary to remember to disable the flags corresponding to any resource that might not be present in the window (menu, icon, or accelerator table), which would otherwise return an invalid handle and fail to display the window on the screen.

In some cases it is even possible that there are no logical errors at all in the function's parameters, but that you might have forgotten to set the `WS_VISIBLE` flag that brings about the actual immediate display of the window on the screen.

---

## Displaying a Window

The `WS_VISIBLE` style implies that the window will be displayed at the very moment it has been created. At times, however, it is preferable not to set this flag, and distinguish two different phases by delaying the window's actual display to a later time. In this case you should use *WinShowWindow()* or—even better—*WinSetWindowPos()*:

```
#define INCL_WINWINDOWMGR
BOOL APIENTRY WinShowWindow( HWND hwnd, BOOL fShow) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of a frame window
fShow	Boolean to state the visibility value
<i>Return Value</i>	<i>Description</i>
BOOL	Success/failure of the operation

The effect of *WinShowWindow()* is simply that of displaying or hiding the pixels that make up the image of the window. The second parameter determines whether the window should be exposed (TRUE) or hidden (FALSE). Often, it is convenient to have the two phases, creation and display, occur at the same time—to this end you just include the flag `WS_VISIBLE`. At other times, *WinShowWindow()* is useful because it is able to make a window disappear from the screen.

If you desire to set the newly created window to a given position, you should use *WinSetWindowPos()*, an API function that adds displaying capabilities, in addition to those of setting the size and position.

```
#define INCL_WINMESSAGEGR
BOOL WINAPI WinSetWindowPos(   HWND hwnd,
                               HWND hwndInsertBehind,
                               LONG x,
                               LONG y,
                               LONG cx,
                               LONG cy,
                               ULONG f1) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of a frame window
hwndInsertBehind	Handle of a top-level window
x	Window lower left corner position on the X axis
y	Window lower left corner position on the Y axis
cx	Window dimension on the X axis
cy	Window dimension on the Y axis
f1	One or more SWP_ flags
<i>Return Value</i>	<i>Description</i>
BOOL	Success/failure of the operation

*WinSetWindowPos()* is easy to use despite being a rather complex function. It is also very flexible, as it can accomplish different requirements at the same time. We have stated before that it is not possible to establish a window's size and position at the time it is created. *WinSetWindowPos()* overcomes this problem and allows a window to be displayed at a position that is different from the default one chosen by the system.

The first of the two window handles taken by this function identifies which handle you are referring to—in our case, it would be the window newly created with *WinCreateStdWindow()*. The second handle establishes the display depth position of the window relative to all other windows that might already be present on the screen. PM stacks the windows on the screen in their order of creation. The window created last will appear as the first. Naturally, the user's interactions might vary the relative onscreen positions of the windows.

**Table 2.9 The HWND\_xxx Defines: Only HWND\_TOP and HWND\_BOTTOM Can Be Used with *WinSetWindowPos()***

<i>Define</i>	<i>Value</i>	<i>Description</i>
HWND_DESKTOP	(HWND)1	Handle of the desktop window.
HWND_OBJECT	(HWND)2	Handle of the base object window.
HWND_TOP	(HWND)3	Handle of the topmost window in the list.
HWND_BOTTOM	(HWND)4	Handle of the window at the bottom of the list.
HWND_THREADCAPTURE	(HWND)5	Definition to identify all windows in a thread.

The second parameter of *WinSetWindowPos()* can take the value of HWND\_BOTTOM and/or HWND\_TOP, in order to arrange the window as the last or first window among all active windows at the same hierarchical level. You can be yet more selective by indicating the handle of a known window in order to establish that the selected window should appear immediately behind the one indicated. Table 2.9 summarizes all the HWND\_ definitions.

In order to activate the handle given as the second parameter to the function, it is necessary to set the SWP\_ZORDER flag among those flags that characterize this function.

To set the window's position and size, use the two LONG pairs (x, y) and (cx, cy). The first pair will tell the system the position of the lower left-hand corner of the window, while the second pair denotes the size of the window in the direction of the x and y axis. All these values have to be expressed in screen-relative coordinates, and thus are dependent on the screen adapter being used or on what is actually specified in the system's configuration file. In order to force the setting of a specific window position and size, you have to set respectively the SWP\_MOVE and the SWP\_SIZE flags. Table 2.10 describes the actions performed by each flag available to the *WinSetWindowPos()* function.

**Table 2.10 List of Flags Acceptable by the *WinSetWindowPos()* Function**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
SWP_SIZE	0x0001	Indicates the change of the window's size by specifying a pair of cx and cy values different from those previously assigned to the window.
SWP_MOVE	0x0002	Allows modification of the (x, y) coordinate of the window's upper left-hand corner position.

*(continued)*

**Table 2.10 (Continued)**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
SWP_ZORDER	0x0004	Modifies the window's position with respect to the window present on the screen at that moment.
SWP_SHOW	0x0008	Displays the window.
SWP_HIDE	0x0010	Forces the window to disappear from the screen, making it invisible.
SWP_NOREDRAW	0x0020	Does not display any changes made to the window.
SWP_NOADJUST	0x0040	Does not allow the WM_ADJUSTWINDOWPOS message that normally follows the usage of <i>WinSetWindowPos()</i> to be sent, and thus will not let the window refresh its position on the screen.
SWP_ACTIVATE	0x0080	The window is activated. The titlebar will be colored with the activation color (blue in the default color settings) and the input focus will be concentrated on the window.
SWP_DEACTIVATE	0x0100	Deactivates the window, provided it was previously active.
SWP_MINIMIZE	0x0400	This flag indicates that the window has to be minimized. In OS/2 2.x this behavior usually makes it disappear.
SWP_MAXIMIZE	0x0800	This flag indicates that the window has to maximize its pixel extent on the screen, so that it will correspond to the whole screen area.
SWP_RESTORE	0x1000	Restores the window's size and position to those it held before being maximized or minimized.
SWP_FOCUSACTIVATE	0x2000	Specifies that the focus is about to be transferred to the frame window. In this way the application that is processing the WM_ADJUSTWINDOWPOS message can determine whether or not that message has been produced by a change of focus.
SWP_FOCUSDEACTIVATE	0x4000	Indicates that the frame window is about to lose the focus.

**Table 2.11** Messages Generated by *WinSetWindowPos()*

<i>Message</i>	<i>Description</i>
WM_CALCVALIDRECTS	Issued so that you can determine the window's size in order to be able later to restore its original size, if changed.
WM_SIZE	Issued if SWP_SIZE is set: contains in mp1 the updated size of the window.
WM_MOVE	Issued whenever the window's on-screen size is changed, but only if the window belongs to a class for which the CS_MOVENOTIFY has been set.
WM_ACTIVATE	Issued whenever there is a focus shift from one window to another.
WM_ADJUSTWINDOWPOS	This message contains the dimensions suggested with the x, y, cx, and cy parameters to change the window's size. If the message is intercepted, it will allow you to make additional changes to the window's size. If the flag SWP_NOADJUST has been set, then this message is not issued.

The action of *WinSetWindowPos()* affects the behavior of the application by triggering various messages according to the flags that have been set; the action is in turn affected by some features of the window being positioned. If the CS\_SIZEREDRAW style has been set, then the use of *WinSetWindowPos()* will automatically make invalid the whole client window of the window, and that will force an immediate on screen redraw of the window's contents.

The use of *WinSetWindowPos()* also causes the automatic sending of the messages indicated in Table 2.11, addressed to the window procedure of the window indicated by the function's *hwnd* parameter.

---

## The Message Loop

In Chapter 1 we stressed the mechanism that governs multitasking in PM; it is no longer preemptive like in OS/2, but simply event driven in order to carry out the user's intentions. Every application must thus constantly adapt its own behavior to the choices made by the user, an operation that is carried out by the system's issuing one or more messages directly to the application's message queue. Thus, every program will examine the contents of its message queue, trying to retrieve as quickly as possible all relevant messages (in Chapter 3 we will examine in depth the nature of the message-based multitasking in PM).

For this reason, you must always insert into *main()* the message loop—a set of functions whose purpose is that of monitoring the message queue, interpreting the messages found, and then dispatching them to the appropriate window procedure. A few lines of code comprise the entire mechanism of PM applications. The principal function of any message loop is *WinGetMsg()*:

```
#define INCL_WINMESSAGEMGR
BOOL APIENTRY WinGetMsg(  HAB hab,
                          PQMSG pqmsg,
                          HWND hwndFilter,
                          ULONG msgFilterFirst,
                          ULONG msgFilterLast) ;
```

<i>Parameter</i>	<i>Description</i>
hab	Handle to the anchor block
pqmsg	Address of a QMSG structure
hwndFilter	Handle of the window for which the messages are to be retrieved
msgFilterFirst	Lower (numerical) limit of the message range to be filtered
msgFilterLast	Upper (numerical) limit of the message range to be filtered
<i>Return Value</i>	<i>Description</i>
BOOL	Success/failure of the operation

Apart from the anchor block handle, the remaining four parameters perform a selective action on the messages found in the queue. The address of a QMSG structure identifies the memory area where the information packet associated with every message found in the queue is stored. The seven members of the QMSG structure fully describe all pieces of information needed by the application in order to qualify the message, the receiver, and the event that has generated it:

```
#define INCL_WINMESSAGEMGR
typedef struct _QMSG
{ // qmsg
  HWND hwnd ;
  ULONG msg ;
  MPARAM mp1 ;
  MPARAM mp2 ;
  ULONG time ;
  POINTL pt1 ;
  ULONG reserved ;
} QMSG ;
```

The *hwnd* member corresponds to the window that should receive the message. In *msg* you will find the actual message, as it is described in *PMWIN.H*. A message is simply a number that corresponds to a symbol defined in *PMWIN.H*. For instance, the message that indicates the pressing of a keyboard key is *WM\_CHAR*, and it is defined:

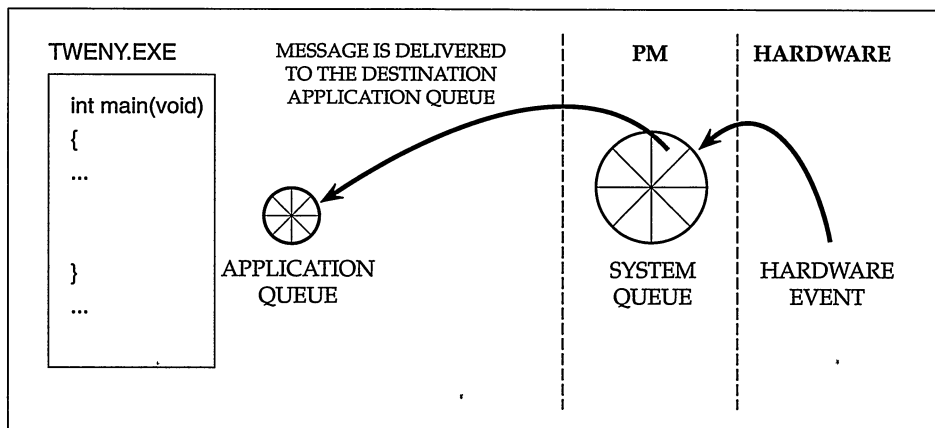
```
...
#define WM_CHAR  0x007a
...
```

The two MPARAM parameters are the true memory area in which the issued message's associated information is stored. The ULONG time indicates when the message actually took place, expressed as the number of milliseconds since the system was turned on. The pt1 structure contains the mouse pointer's position on the screen when the message was generated. This information provides a clear picture of what happened in the system at the moment when the message was produced.

Every hardware event is transformed by the system into one or more messages posted into the queue of the appropriate application. For instance, the enlargement of a window's surface causes the system to generate a series of messages to notify a changing of dimensions to the correct window. Figure 2.8 summarizes this mechanism.

The third parameter of *WinGetMsg()*, the handle of the filter window, will usually take the value of NULLHANDLE. This indicates that the function should capture any message addressed to any window of the application. Should you want to focus on a particular window and its possible children, you have to specify its handle as it was produced by *WinCreateStdWindow()*. Such a change to the message loop also implies that a PM application can have more than one loop for retrieving messages from the queue (one for each PM thread).

Finally, the last two parameters determine which messages should be retrieved from the queue by indicating the first and the last one in a range of possible messages. These parameters take the value of zero to indicate that the request has been made to retrieve a message from the queue. PMWIN.H contains several constants that define message ranges and that can be used in *WinGetMsg()*. For instance, to retrieve mouse messages only, you can specify respectively WM\_MOUSEFIRST and WM\_MOUSELAST corresponding to the parameters msgFilterFirst and msgFilterLast. Similarly, WM\_DDE\_FIRST and WM\_DDE\_LAST allow you to identify only those messages belonging to the DDE communication protocol.



**Figure 2.8** A hardware event like the movement of the mouse pointer is transformed by the system into a message posted to the message queue of the application involved in the operation.



Once a message has been retrieved, the only thing that has to be done is to post it for further processing to the appropriate window procedure. The API of PM provides *WinDispatchMsg()* for managing this chore.

```
#define INCL_WINMESSAGEGR
MRESULT APIENTRY WinDispatchMsg( HAB hab, PQMSG pqmsg ) ;
```

<i>Parameter</i>	<i>Description</i>
hab	Handle to the anchor block
pqmsg	Address of a QMSG structure
<i>Return Value</i>	<i>Description</i>
MRESULT	Window procedure return value

*WinDispatchMsg()* performs its action based on two parameters alone without any indication of which window procedure is the addressee.

When you register a window class, you have to indicate a window procedure for that class. This was accomplished by providing the name of a function. In the C language, this kind of identifier actually corresponds to a pointer that holds the address of the code segment where the function body is actually stored.

Therefore, when you register a window class you are also telling the system where it can find the associated window procedure. *WinDispatchMsg()* needs to reach this window procedure. It will take advantage of the first member of the QMSG structure, that is, the handle of the receiver window. Through this window handle, the system is able to discover the class to which it belongs, and thus the name of the window procedure that was established at registration time. It can then transfer execution precisely to that window procedure. With such a scheme, the message loop becomes an extremely flexible tool, especially if you consider that an application can have several window classes, each one of which possesses its own window procedure. Calling a window procedure directly would mean being able to execute no more than one window class registration alone.

As we will discover in Chapter 4, where all issues related to windowing will be treated in depth, the information pertaining to the window procedure is stored not only at the class level, but even within each single window. Therefore, every window, or rather every *hwnd*, contains in its own reserved memory space this piece of information that is really critical for the *WinDispatchMsg()* function and menu other window-related tasks.

```
...
while( WinGetMsg( hab, &qmsg, NULLHANDLE, 0L, 0L)
    WinDispatchMsg( hab, &qmsg) ;
...

```

Once we get this far in the code of *main()*, the window is displayed on-screen ready to interact with the user and to answer any actions.

## *Living in the Message Loop*

So far we have examined the *main()* function and performed all preliminary and preparatory action for a PM action. The message loop, on the other hand, is the engine that powers the event-driven multitasking so typical of the PM screen group, which is able to suspend the execution of an application when any kind of message is detected in the message queue.

The value returned by *WinGetMsg()* determines the runtime duration of the *while* loop. Whatever message is extracted from the queue will produce a nonzero value, with the exception of *WM\_QUIT* that breaks the loop and signals the end of the program's existence, as it will no longer be able to access the message queue, its primary and vital source of information.

The system immediately issues a *WM\_QUIT* message whenever a window is closed through a double-click on the titlebar's icon. The application achieves this same result alone by issuing the *WM\_QUIT* message directly into its own queue.

## *Execution Termination*

Once you have exited from the message loop, you must terminate the application by destroying all resources that have been created in the code. The message queue is destroyed with a special function, *WinDestroyMsgQueue()*:

```
#define INCL_WINMESSAGEGR
BOOL APIENTRY WinDestroyMsgQueue( HMQ hmq ) ;
```

<i>Parameter</i>	<i>Description</i>
hmq	Application's message queue handle
<i>Return Value</i>	<i>Description</i>
BOOL	Success/failure of the operation

To destroy the window corresponding to the application, you have to use *WinDestroyWindow()*:

```
#define INCL_WINWINDOWMGR
BOOL APIENTRY WinDestroyWindow( HWND hwnd ) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Valid window handle
<i>Return Value</i>	<i>Description</i>
BOOL	Success/failure of the operation

To terminate program execution, you can call *WinTerminate()*, which has the effect of canceling the handle to the application's anchor block.

```
#define INCL_WINWINDOWMGR
BOOL APIENTRY WinTerminate( HAB hab ) ;
```



```

while( WinGetMsg( hab, &qmsg, NULL, 0, 0))
    WinDispatchMsg( hab, &qmsg) ;

WinDestroyMsgQueue( hmq) ;
WinDestroyWindow( hwndFrame) ;
WinTerminate( hab) ;

return 0 ;
}

```

---

## The Window Procedure

The term *window procedure* identifies a function that performs a very special duty within a PM application. A typical PM executable always requires a piece of code specifically designed for the window procedure, although the actual name assigned is determined by the programmer. When defining the function prototypes present in the code, we have assigned to the window procedure the conventional name *ClientWndProc()*. This function will be called by *WinDispatchMsg()* according to the mechanism described every time that *WinGetMsg()* retrieves a message belonging to the class associated with this window procedure from the message queue. Since our first PM application will have only one window belonging to a class that has been registered before entering the message loop, any interaction with the window on the screen will generate a message posted to *ClientWndProc()*. This is a somewhat simplified explanation, because true applications will ordinarily register more than one class. The parameters of a generic window procedure correspond to the first four members of a QMSG structure.

```

...
MRESULT EXPENTRY ClientWndProc(  HWND hwnd,
                                  ULONG msg,
                                  MPARAM mp1,
                                  MPARAM mp2)
{
    ...
}

```

In general, the body of a window procedure is quite complex, as it has to deal with the processing of several different messages. In this first example, we limit our efforts to simply returning a value from the function, as required by its prototype. You can use the *WinDefWindowProc()* function, the default window procedure that is part of the API of PM, to accomplish this. The parameters of this function are exactly the same as those of a window procedure, since its purpose is that of providing a default processing for all those messages that are not captured directly in the function. By setting:

```

return WinDefWindowProc( hwnd, msg, mp1, mp2) ;

```

you can force the default processing provided by the system on any message that reaches the window procedure. The role played by the window procedure is critical in the design of a PM application, as all messages generated by the interaction between the user and a window are passed to this kind of function, within which the true logic of the program is implemented. The *main()* function is used almost exclusively for setting up the application. Once the set-up phase is over, the message loop-window procedure takes over the real work and characterizes each PM application.

Let's summarize the main points that pertain to the framework of a PM application:

- Each window procedure present in the code will always be an EXPENTRY function and will be exported to the module definition file correspondingly to the EXPORTS directive.
- The number of window procedures is always equal to the number of window classes registered, provided that you don't reference a class like CS\_PUBLIC, or that more classes share one window procedure.

## A Sample Application

Let's now examine the entire code that constitutes our first PM application. In Listing 2.7 you will find the make file, in Listing 2.8 the module definition file, and finally in Listing 2.9, the source code. The header file and the resource file are not listed, because they are not necessary in a simple program.

### Listing 2.7 The Make File MACHINE.MAK

---

```

PROJ = machine
PROJFILE = machine.mak
DEBUG = 1

PWBRMAKE = pwbrmake
NMAKEBSC1 = set
NMAKEBSC2 = nmake
LINKER = link
ILINK = ilink
LRF = echo > NUL
BIND = bind
RC = rc
IMPLIB = implib
LFLAGS_G = /STACK:8192 /NOI /BATCH
LFLAGS_D = /CO /INC /FAR /PACKC /PACKD /PMTYPE:PM
LFLAGS_R = /EXE /FAR /PACKC /PACKD /PMTYPE:PM
MAPFILE_D = NUL
MAPFILE_R = NUL
CC = cl
CFLAGS_G = /W3 /G2 /Zp /BATCH
CFLAGS_D = /qc /Gs /Gi$(PROJ).mdt /Zr /Zi /Od
CFLAGS_R = /Ot /Oi /O1 /Oe /Og /Gs

```

```

ASM = masm
AFLAGS_G = /Mx /T
AFLAGS_D = /Zi

OBJS = machine.obj
SBRS = machine.sbr

all: $(PROJ).exe

.SUFFIXES: .c .sbr .obj
machine.obj : machine.c
machine.sbr : machine.c

$(PROJ).bsc : $(SBRS)
    $(PWBRMAKE) @<<
$(BRFLAGS) $(SBRS)
<<

$(PROJ).exe : $(OBJS)
!IF $(DEBUG)
    $(LRF) @<<$(PROJ).lrf
$(RT_OBJS: = +^
) $(OBJS: = +^
)
$@
$(MAPFILE_D)
$(LLIBS_G: = +^
) +
$(LLIBS_D: = +^
) +
$(LIBS: = +^
)
$(DEF_FILE) $(LFLAGS_G) $(LFLAGS_D);
<<
!ELSE
    $(LRF) @<<$(PROJ).lrf
$(RT_OBJS: = +^
) $(OBJS: = +^
)
$@
$(MAPFILE_R)
$(LLIBS_G: = +^
) +
$(LLIBS_R: = +^
) +
$(LIBS: = +^
)
$(DEF_FILE) $(LFLAGS_G) $(LFLAGS_R);
<<

```

```

!ENDIF
!IF $(DEBUG)
    $(LINK) -a -e "$(LINKER) @$(PROJ).lrf" @$
!ELSE
    $(LINKER) @$(PROJ).lrf
!ENDIF

.c.sbr :
!IF $(DEBUG)
    $(CC) /Zs $(CFLAGS_G) $(CFLAGS_D) /FR$@ $<
!ELSE
    $(CC) /Zs $(CFLAGS_G) $(CFLAGS_R) /FR$@ $<
!ENDIF

.c.obj :
!IF $(DEBUG)
    $(CC) /c $(CFLAGS_G) $(CFLAGS_D) /Fo$@ $<
!ELSE
    $(CC) /c $(CFLAGS_G) $(CFLAGS_R) /Fo$@ $<
!ENDIF

run: $(PROJ).exe
    $(PROJ).exe $(RUNFLAGS)

debug: $(PROJ).exe
    CVP $(CVFLAGS) $(PROJ).exe $(RUNFLAGS)

```

### Listing 2.8 The Module Definition File MACHINE.DEF

---

```

;-----
; MACHINE.DEF module definition file
;-----

NAME        MACHINE
DESCRIPTION 'Stefano Maruzzi, 1993'
PROTMODE
HEAPSIZE    8192
STACKSIZE   8192
EXPORTS
    ClientWndProc

```

### Listing 2.9 The Source Code File MACHINE.C

---

```

// MACHINE.C - A basic PM application
// Listing 02-09

// Stefano Maruzzi 1993

#define INCL_WIN

```

## 70 OS/2 2.1 Workplace Shell Programming

```
#include <os2.h>

// function prototypes
int main( void) ;
MRESULT EXPENTRY ClientWndProc( HWND hwnd, ULONG msg, MPARAM mp1,
MPARAM mp2) ;

int main( void)
{
    HAB hab ;
    HMQ hmq ;
    HWND hwndFrame, hwndClient ;
    QMSG qmsg ;
    ULONG flFrameFlags = FCF_STANDARD & ~FCF_MENU & ~FCF_ICON &
        ~FCF_ACCELTABLE ;
    CHAR szClassName[] = "Machine" ;
    CHAR szWindowTitle[] = "Basic PM application" ;

    hab = WinInitialize( 0) ;
    hmq = WinCreateMsgQueue( hab, 0L) ;

    WinRegisterClass( hab, szClassName,
        ClientWndProc,
        CS_SIZEREDRAW, 0) ;

    hwndFrame = WinCreateStdWindow( HWND_DESKTOP,
        0L,
        &flFrameFlags,
        szClassName,
        szWindowTitle,
        0L,
        NULLHANDLE,
        0L,
        &hwndClient) ;

    WinShowWindow( hwndFrame, TRUE) ;

    while( WinGetMsg( hab, &qmsg, NULLHANDLE, 0L, 0L))
        WinDispatchMsg( hab, &qmsg) ;

    WinDestroyWindow( hwndFrame) ;
    WinDestroyMsgQueue( hmq) ;
    WinTerminate( hab) ;

    return 0L ;
}

MRESULT EXPENTRY ClientWndProc( HWND hwnd,
    ULONG msg,
    MPARAM mp1,
    MPARAM mp2)
{
```



```

switch( msg)
{
    default:
        break;
}
return WinDefWindowProc( hwnd, msg, mp1, mp2) ;
}

```

## Some Alternate Solutions and Enhancements

In Listing 2.9, showing the C source code, the style `WS_VISIBLE` has been set so that the window will be displayed automatically during its creation phase. An alternate solution would be inserting in the code, after the call to `WinCreateStdWindow()`, a call to `WinShowWindow()`, specifying the boolean value of `TRUE`.

An even more interesting solution is to use `WinSetWindowPos()`, allowing a much greater degree of flexibility than the two preceding solutions.

In the first case, the code of `main()` shown in Listing 2.6 would be changed as follows:

```

...
hwndFrame = WinCreateStdWindow(  HWND_DESKTOP,
                                OL,
                                &flFrameFlags,
                                szClassName,
                                szWindowTitle,
                                OL,
                                NULLHANDLE,
                                OL,
                                &hwndClient) ;

WinShowWindow( hwndFrame, TRUE) ;

while( WinGetMsg( hab, &qmsg, NULLHANDLE, 0, 0))
    WinDispatchMsg( hab, &qmsg) ;
...

while with WinSetWindowPos() things are completely different:

...
hwndFrame = WinCreateStdWindow(  HWND_DESKTOP,
                                OL,
                                &flFrameFlags,
                                szClassName,
                                szWindowTitle,
                                OL,
                                NULLHANDLE,
                                OL,
                                &hwndClient) ;

WinSetWindowPos( hwndFrame, HWND_TOP, OL, OL, OL, OL, SWP_SHOW) ;

while( WinGetMsg( hab, &qmsg, NULLHANDLE, OL, OL))
    WinDispatchMsg( hab, &qmsg) ;
...

```

In the former example, both the position as well as the size of the window remain unchanged. This code fragment works on the assumption that the `FCF_SHELLPOSITION` flag has been set. To set the window at a specific position when it is displayed, you must specify pixel coordinates relative to the screen's physical dimensions, and also you must add the flags `SWP_SIZE` and `SWP_MOVE` to `WinSetWindowPost()`, which informs the function that it will be responsible for performing those two actions. In this case, it is suitable, but not necessary, to disable the `FCF_SHELLPOSITION` flag, since its purpose is that of establishing a random position on the screen for the window.

## The Look of Windows

Figure 2.9 shows how the `MACHINE` application would appear at the center of the PM screen group. The client window has exactly the same color as the desktop window. A real-world situation might be very different, though, as is shown in Figure 2.10.

Some objects of WPS appear inside the client window of the `MACHINE` application. You can think of this as if the pixels of the client window were transparent, thus allowing you to see the underlying objects "through" the window. Figure 2.10 shows an unacceptable situation, since now there appear to be two OS/2 System folders in the desktop.

Clearly this behavior of `MACHINE` is abnormal. It occurs because the contents of the client window are not continuously refreshed and its client window is transparent. The pixels of WPS have "exposed" the client window, without any means to refresh it. To solve this sort of problem, it is necessary to master the painting techniques of PM, a subject that will be described in Chapter 3.

## The Application Title

When the fifth parameter of `WinCreateStdWindow()` is set to `NULL`, the titlebar of the window shows the name of the project containing the application, provided the `FCF_TASKLIST` flag is set. Without `FCF_TASKLIST` the window would have no title whatsoever. The change or assignment of the titlebar can take place at a different time than when the window was created by using the `WinSetWindowText()` function:

```
#define INCL_WINMESSAGEMGR
BOOL WINAPI WinSetWindowText( HWND hwnd, PSZ pszText );
```

<i>Parameter</i>	<i>Description</i>
hwnd	Window handle: usually a frame window or a window of a predefined class
pszText	String of text shown in the application titlebar or in a control window
<i>Return Value</i>	<i>Description</i>
BOOL	Success/failure of the operation

It is necessary to be careful with the value of the first parameter—it has to be the handle of the frame window, not the handle of the client window. As we will see in the following chapters, the use of `WinSetWindowText()` is very convenient even for those windows that belong to predefined classes.

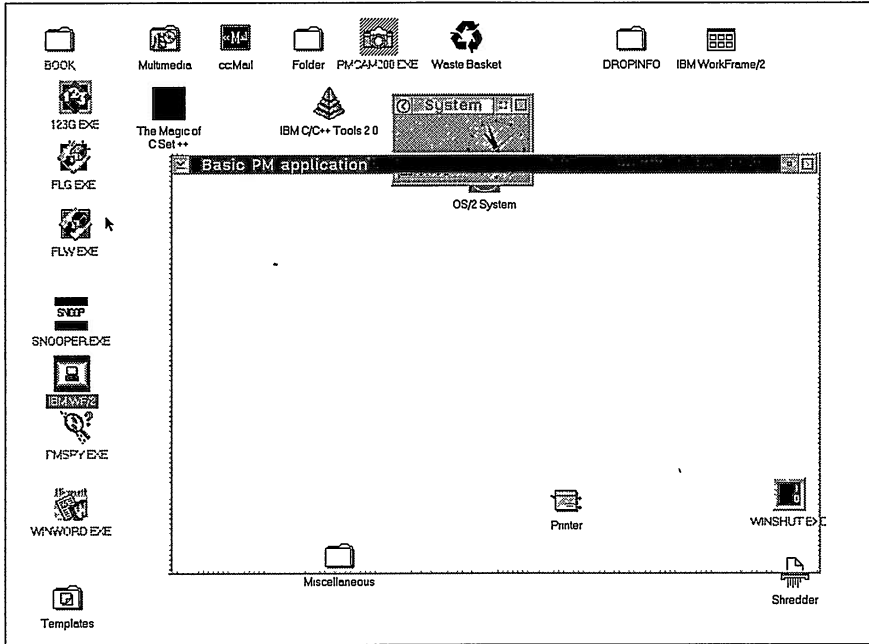


Figure 2.9 Inside the client window of MACHINE you can see portions of other applications.

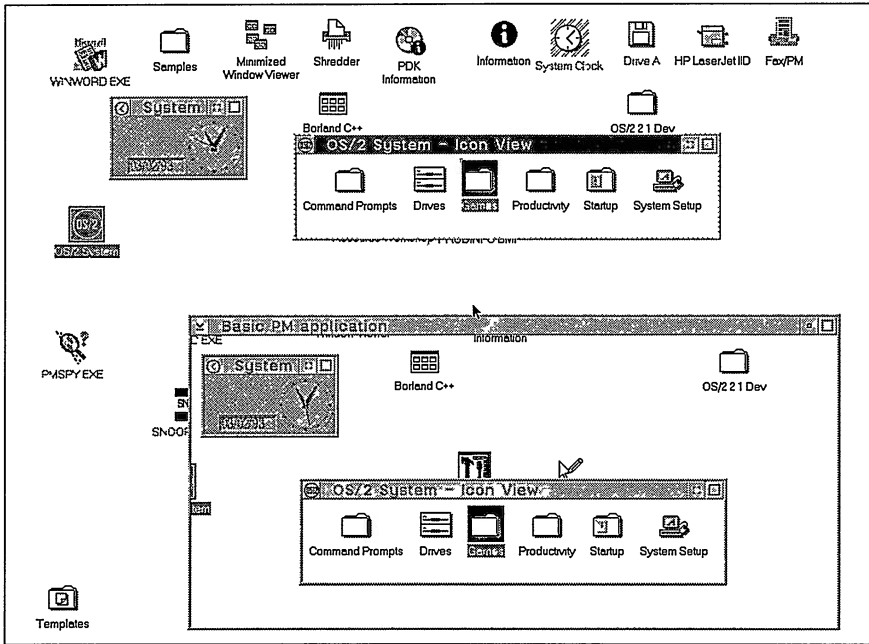


Figure 2.10 A portion of OS/2 System has been frozen in the client window of MACHINE.



# Messages

The first PM application that we designed in the previous chapter allowed us to define the basic structure or backbone of a program. As we have stated, it is the window procedure that takes on the central role in the code, receiving all messages issued by the system or generated by other applications and addressed to a certain window. The window procedure in Listing 2.9 is very simple, and only returns a value after performing the default processing for any message it might receive. More often, however, a window procedure will contain code fragments designed by the programmer devised for the specific processing needed for the messages being sent. The structure of a window procedure can be imagined as a kind of sieve that filters some messages and lets others flow directly to a *WinDefWindowProc()*. So a reasonable question is: Which messages should be caught and which should be sent to *WinDefWindowProc()*? What criteria should be applied? There is no precise answer to these questions, because it all depends on the developer, and on how well the developer knows the PM's API functions and the strategies followed in writing the code.

One criterion that will be followed in all applications presented in this book is that of always passing on any message received by the window procedure to *WinDefWindowProc()*, even if these messages have been caught inside the procedure's body. In this way, any message sent to a window will at least be subject to the system's default processing, and thus will avoid any kind of side effect in the application's working. Very seldom will a message be prohibited from reaching a *WinDefWindowProc()*. Figure 3.1 sketches the action performed by a window procedure on the message flow generated by PM or by other PM applications.

In order to perform this message capture, you have to act on the second parameter of the window procedure, the `ULONG` type identifier that is commonly indicated with the name `msg`. It is a message generated as a consequence of interaction with a window. Naturally, the four specific identifiers of a window procedure can have different names; but a convention that is almost universally followed will invariably use the labels `hwnd`, `msg`, `mp1`, and `mp2`. This happens both in the sample listings of the Toolkit as well as in the articles that appear in programming magazines.

The `msg` identifier contains the numeric value of the message that, at that very moment, is being passed to the window procedure. In `PMWIN.H`, this value is defined by a text string so that it is easier to deal with it. To do this a `#define` preprocessor directive is used:

```

MRESULT EXPENTRY ClientWndProc(  HWND hwnd,
                                  ULONG msg,
                                  MPARAM mp1,
                                  MPARAM mp2)
{
    switch( msg)
    {
        case WM_XXX:
            ...
            break ;

        case WM_YYY:
            ...
            break ;

        default:
            ...
            break ;
    }
    return WinDefWindowProc( hwnd, msg, mp1, mp2) ;
}

```

**Figure 3.1** The message flow in a PM application.

```

...
#define WM_CREATE 0x0001
...

```

Appendix C lists all WM\_ messages currently available in OS/2 2.1. Therefore, to discover which message has been sent to a window procedure, just examine the contents of *msg* and differentiate the application's behavior according to what condition is assessed.

However, in many cases, this kind of information is available directly in the message loop, before you ever proceed to dispatching the message with *WinDispatchMsg()*. In such a context, the *WinGetMsg()* function needs the address of a QMSG structure that contains among its seven members a ULONG identifier called *msg*—the message that will eventually be received by the window procedure.

Inside a window procedure, you can implement customized processing for the various messages that need to affect the application. Catching all messages passing by would not be useful and would be a nuisance for the programmer. The typical organization of a window procedure will therefore bear a *switch* statement on the *msg* identifier, with one or more *case* conditions. Each *case* branch corresponds to a separate message that you intend to process. Due to the great number of messages that are present in PM (Appendix C lists only those referring to on-screen window management, introduced by the WM\_ prefix), it might seem impossible to settle on

criteria for deciding which message should be processed directly by the application in its window procedure and which should be passed on directly to *WinDefWindowProc()*. Practice and experience will help you make informed decisions on all messages generated by the system when managing windows (WM\_). The diverse categories of messages that are part of PM's API are listed in Table 3.1.

Of all these messages, only the WM\_ messages actually reach a window procedure, in direct response to what the user does to the application's window or by the functions and operations performed internally by the code. The *drag&drop* messages are something of a novelty in OS/2 2.x, and are used mainly for dealing with the user's selecting, dragging, and dropping objects between windows (Chapter 12).

**Table 3.1 The Various Categories of Messages in PM's API**

<i>Prefix</i>	<i>Description</i>
WM_	Window message: a message that refers to managing the position, output, and size of a window on the screen.
EM_	Class WC_ENTRYFIELD: a message for managing the user's input of alphanumeric strings in an entry field window.
BM_	Class WC_BUTTON: a message for controlling push buttons, check boxes, radio buttons, and other kinds of buttons.
LM_	Class WC_LISTBOX: a message for dealing with all activities involved in using a list box.
MM_	Class WC_MENU: a message for inserting new menu and new options into existing menus.
SBM_	Class WC_SCROLLBAR: a message for dealing with the movements of a horizontal or vertical scroll bar's thumb.
TBM_	Class WC_TITLEBAR: a message to set or retrieve any text contained in an application's titlebar.
BKM_	Class WC_NOTEBOOK: a message to customize the look and behavior of an object that looks like a notebook.
CM_	Class WC_CONTAINER: a message for defining the contents and appearance of a folder.
SLM_	Class WC_SLIDER: a message for customizing the behavior and appearance of a slider.
SPBM_	Class WC_SPINBUTTON: a message for defining the range, text alignment, and contents of a spin button.
VM_	Class WC_VALUESET: a message used to query and set the cells in a valueset.
DM_	Drag message: a message generated in response to a user's drag & drop actions.

The other messages are sent from the applications to windows belonging to the predefined PM classes (Chapter 7). This means that these messages will never be caught by a window procedure of a class registered by the program because they will never reach it. In fact, their destinations are the respective window procedures of the predefined classes. The only way to allow the developer's code to deal directly with messages, such as `LM_` or `VM_`, is through subclassing. This powerful technique, described in Chapter 10, lets you modify the natural flow of messages addressed to a window of a predefined class.

---

## Painting

Unlike a character-based user interface program, a PM application displays its output in a unique way. A *full screen* executable simply assumes that all of the screen's 80 columns and 25 rows are at its complete disposal. In the case of PM, this is not true, because it is a graphical user interface completely disengaged from the very concept of rows and columns. Furthermore, since PM is a multitasking environment, it has to be able to deal simultaneously with several applications occupying different parts of the screen, giving rise to situations where windows will overlap. The logic that governs output is generally referred to as *painting*.

Among the structural components of a window, only the client window is conventionally used by the application to perform its drawing and text display operations, implementing a kind of dynamic and iterative form of output. In other words, it is possible to regenerate the output within an application's client window at any time. There are a number of conditions that trigger a refresh of the client window's contents. The most common are those due to some resizing operation of a window's output area or to some kind of interaction between different windows. This last category is represented, for example, by windows overlapping each other or by closely related windows (parent/children). Both situations produce a variation in the quantity of pixels available to an application (window) for its output. In order to notify a window that it is time to refresh its output, the system posts into the application's message queue—the queue created by calling `WinCreateMsgQueue()`—the `WM_PAINT` message, whenever that might be necessary. The application retrieves the `WM_PAINT` message from the queue while in its message loop, and then sends it directly to the appropriate window procedure—that is, to the window procedure of the class to which the window in need of regeneration belongs.

All of the application's output logic is collected in the case `WM_PAINT` statement that the programmer writes in the `switch` block based on the `msg` identifier. It is in this portion of the code that all of the window's output management functions are called. Therefore, a generic window procedure will *always* contain a case `WM_PAINT` condition, and that code fragment takes care of all activities involved in generating the program's output. Accordingly, we will change our window procedure's framework by admitting a case branch for dealing with all the program's output chores (Listing 3.1).



**Listing 3.1 Processing of a Message in a Window Procedure**

---

```

MRESULT EXPENTRY ClientWndProc(  HWND hwnd,
                                ULONG msg,
                                MPARAM mp1,
                                MPARAM mp2)
{
    switch( msg)
    {
        case WM_PAINT:
        {
            ...
        }
        break ;

        default:
            break ;
    }
    return WinDefWindowProc( hwnd, msg, mp1, mp2) ;
}

```

All messages received in the window procedure in Listing 3.1 enter the `switch` block. All of them, except `WM_PAINT`, are handled directly by the default branch which forwards them to the processing of `WinDefWindowProc()`.

<b>WM_PAINT</b>	<b>0x0023</b>
mp1	Reserved
mp2	Reserved
Return Value	Reserved

The `WM_PAINT` condition must be set in such a way that when this message (0x0023 in `PMWIN.H`) is received, the corresponding piece of code is executed (indicated by the ellipses in Listing 3.1), instead of being passed straight to the default processing in `WinDefWindowProc()`. Actually a `break` statement, as in Listing 3.1, does not produce any functional difference because no concrete action is performed on the intercepted message. However, it does delimit the area in which the programmer can make code changes or additions.

Our exercise will be that of painting the window's background—the client window—in color. This kind of task is usually performed by calling the `GpiErase()` function, which is part of the `GPI` portion of `PM`'s API:

```

#define INCL_GPICONTROL
BOOL APIENTRY GpiErase( HPS hps) ;

```

<b>Parameter</b>	<b>Description</b>
hps	Handle to the presentation space
<b>Return Value</b>	<b>Description</b>
BOOL	Success or failure of the operation

The only parameter required by *GpiErase()* is a handle to a presentation space, a new kind of information, specific to output operations. Another solution is that of using the *WinFillRect()* function that takes on the following syntax:

```
#define INCL_WINWINDOWMGR
BOOL APIENTRY WinFillRect( HPS hps, PRECTL prectl, COLOR clr ) ;
```

<i>Parameter</i>	<i>Description</i>
hps	Handle to a presentation space
prectl	Address of an identifier of type RECTL inside which you must supply the window's dimension
clr	Numeric define corresponding to the selected color
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

After the presentation space's handle, you have the address to a RECTL structure and a code value corresponding to one of the predefined colors used to paint the window background. *WinFillRect()* fills that rectangle with the given color. The function is applicable in different contexts; in this specific case, the rectangle corresponds to the entire surface of the client window. A common method by which to retrieve this data is by calling the *WinQueryWindowRect()* function:

```
#define INCL_WINWINDOWMGR
BOOL APIENTRY WinQueryWindowRect( HWND hwnd, PRECTL prectl ) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window of which you want to know the dimensions
prectl	Address of a RECTL structure inside which the window's dimensions will be inserted
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

*WinQueryWindowRect()* can be used any time you need to know the size of an on-screen window. This function must be given the window's handle and the address to a RECTL identifier. This structure features four LONG members that describe two vertexes of a rectangle: the upper left and the lower right. The purpose of *WinQueryWindowRect()* is to compute the number of pixels taken on the screen by the surface occupied by the window. The RECTL structure, as described in PMWIN.H, looks like this:

```
typedef struct _RECTL
{ // rc
    LONG xLeft ;
    LONG yBottom ;
    LONG xRight ;
    LONG yTop ;
} RECTL ;
```

Once you have this information, you can instruct the system to paint in a given color all pixels enclosed by that rectangle. Like *GpiErase()*, *WinFillRect()* requires as its first parameter a handle to a presentation space. This is a common feature of all PM functions involved in the production of any kind of output. The information contained in a presentation space handle is absolutely critical for PM in order to perform any kind of window output operation, such as displaying a text string, a bitmap, or a drawing.

## Output Techniques in PM

One of PM's distinctive traits is that it allows the development of applications that have a *device-independent* output logic. This means that when you are writing an OS/2 application, any display operation (text or images) is not directly related to the actual features of the hardware adapter installed in the system. Naturally, the software designer has to consider the screen's dimensions in order to position any window appropriately and make it visible. However, it is not necessary to set any specific sizes on the sides of a square to ensure that proportions are kept right on both axes.

This means that the same code transferred to a system with different resolution will continue to operate correctly and adapt itself dynamically to the new situation. Drawing a circle on a Personal Computer equipped with a VGA adapter will achieve the same result if you decide to change to an XGA resolution (obviously, the ratio between the x and y axis, as well as the physical screen area can vary according to the greater resolution that one adapter might have over another). All painting activities performed by PM follow some special rules, shown in Figure 3.2.

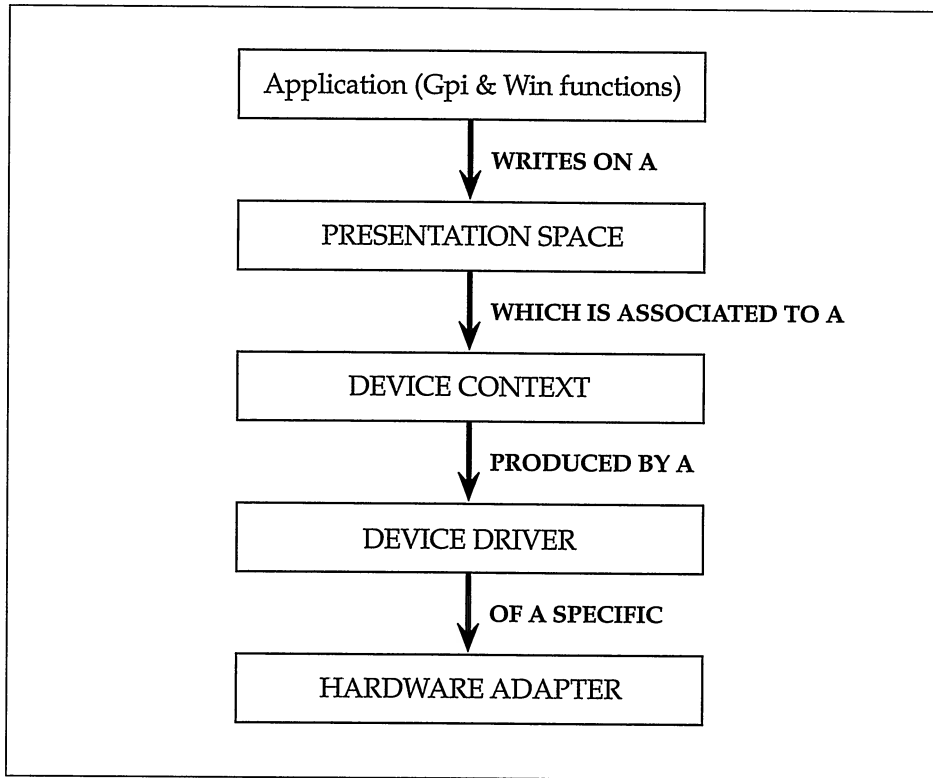
Examine Figure 3.2 starting from the bottom. The video adapter installed in the system is handled by OS/2 through specific *device drivers*, which are identified in the CONFIG.SYS configuration file. The following lines, extracted from CONFIG.SYS, show how—in this case—the system is set up to take advantage of a video adapter capable of the XGA standard.

```
...
DEVICE=C:\OS2\XGARINGO.SYS
DEVICE=C:\OS2\XGA.SYS
SET VIDEO_DEVICES=VIO_XGA
SET VIO_XGA=DEVICE(BVHVGA,BVHXGA)
...
```

The DEVINFO directive in CONFIG.SYS is employed to prepare a device for using a certain code page. In this case, the VGA resolution and the file VIOTBL.DCP are selected for managing the code page.

```
...
DEVINFO=SCR,VGA,C:\OS2\VIOTBL.DCP
...
```

The actual device drivers that make the system hardware independent are internal system items, and can be thought of as software tools used by OS/2 for interacting



**Figure 3.2** Functional scheme of output operations performed by a PM application.

and communicating with the underlying hardware. The greater the proficiency in exploiting the hardware's potential performance, the greater will the advantage be for the application in terms of available resolution and colors. The information returned by a device driver is accessible to an application by means of a *device context*. A device context is a set of data stored by the system that describes the functional and structural features of some hardware component. Device contexts are not limited to video adapters; they are produced by any kind of device driver that can interact with adapters or other hardware equipment capable of generating output, like printers and plotters.

A device context is strongly related to a hardware device. When an application has access to a device context, it has already established which output device operations will be addressed. Despite its importance, it is not the key element to generate output. As you can see in Figure 3.2, an application's code is only marginally interested in a device context, because all functions regarding any output activity will operate on a *presentation space*. A presentation space is one of the system's many internal structures; it describes the characteristics of the area in which the graphical functions are allowed to operate. The *GPI* functions of PM's API generate their output in a presentation space, which, in turn, is associated with a device context.

To understand the relationship existing between a presentation space and a device context, imagine the first one as the negative film exposed to the images framed by a camera at the very moment a snapshot is taken (action produced by *GPI* and some *WIN* functions). The negative film is useful by itself, but it cannot be considered a true form of output. To obtain the final result, you need to print the image on photographic paper that has its own size, texture, weight, and capability in rendering true colors. The transfer of the potential picture into something tangible is the true output process.

The application, thus, draws on a presentation space, referring to whatever coordinate system, resolution, and unit of measure it pleases. This activity does *not* produce the display of any image, because the presentation space is not directly connected to any physical device in the computer. Once an image has been created in the presentation space, and once the actual media that will display the resulting work has been chosen (screen, printer, plotter, etc.) you only need to associate the presentation space with the appropriate device context. This last point also involves some conversion and a mapping process of the virtual image drawing units defined by the programmer into physical points that can be handled by the device.

### *Distinction between Presentation Space and Device Context*

The advantage of separating a virtual image (PS) from its concrete representation (DC) is to be found in the greater flexibility gained by all of PM's output operations. In fact, you might draw a circle on a generic PS and then associate selectively that PS with three or more distinct device contexts, so as to obtain the rendering on the screen, on a laser printer, or on a plotter.

The adoption of a presentation space, in addition to a device context, is new to the painting model used in MS Windows, where the DC is the only layer between the application and the hardware. The reason for this additional intermediate layer is to make the drawing phase of an output process as independent as possible from its actual display. Without a presentation space, the application has to get a valid reference (a handle) to a device context, and thus decide even before producing any output at all, what the final target media should be. In PM the presence of a presentation space allows the selection of a physical output device to be deferred, and still allows the drawing to be generated. With the behavioral model adopted by MS Window, when you want to reproduce an image on two different devices (for instance, the screen and a printer), you are forced to generate two different instances of the same drawing, one for each device context, and to execute the whole set of drawing functions twice. This can be avoided with the PM distinction of PS and DCs.

---

## The Presentation Spaces

In PM there are three types of presentation spaces: *cached micro PS*, *micro PS*, and *normal PS*. Very often, in the Toolkit's support documentation or in other texts on OS/2 programming, the distinction is limited to the last two categories, since the cached

micro PS is a very simple and fast solution with limited potential. In the following pages we will always use a cached micro PS to solve output problems in the client window, because the examples we will examine perform very limited painting activities. In these cases a cached micro PS is always the best approach.

## The Cached Micro PS

The main advantage of the cached micro PS is that it can provide a handle to a presentation space (hps) that is directly associated to a screen device context. All information needed to perform the output is available at once. The term *cached* refers to the nature of the PS returned. The presentation space is not created by the application, but is predefined in the system. A cached micro PS is automatically associated only with the device context related to the video adapter present in the system. Therefore, it is a simple and optimized tool for performing output operations directly in the application's window on the screen. To obtain a cached micro PS from PM, you need to call the *WinGetPS()* function:

```
#define INCL_WINWINDOWMGR
HPS APIENTRY WinGetPS( HWND hwnd ) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window of which you want to get a presentation space
<i>Return Value</i>	<i>Description</i>
HPS	Handle of a presentation space

Once the handle of a window has been specified, the system automatically provides the application with a presentation space valid for generating simple output.

Due to the cached nature of this object, it is necessary to release the presentation space once you have finished using it. Returning a PS handle to PM makes it accessible to other applications. The operation is performed with *WinReleasePS()* which invalidates the handle referring to the presentation space previously obtained through *WinGetPS()*.

```
#define INCL_WINWINDOWMGR
BOOL APIENTRY WinReleasePS( HPS hps ) :
```

<i>Parameter</i>	<i>Description</i>
hps	Handle of a presentation space
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

A cached micro PS can be returned to the system at any time during the application's execution. In PM there is no need to request and release a cached micro PS within the code fragment that deals with a single message. It is possible to obtain a handle to a presentation space correspondingly to a generic WM\_ message, and to release it back to the system when you are processing another message. Naturally, this implies that

the hps identifier be declared with a static storage class. In general, however, it is advisable to use the functions *WinGetPS()* and *WinReleasePS()* entirely within the code fragment dealing with a single message. This helps you write understandable and easy to maintain code.

```

...
case WM_XXX:
{
    HPS hps ;

    hps = WinGetPS( hwnd ) ;
    ...
    WinReleasePS( hps ) ;
}
...

```

The pair of functions *WinGetPS()* and *WinReleasePS()* can be employed in the code fragments dealing with any message inside a window procedure, with the exception of WM\_PAINT. In this case you must use the *WinBeginPaint()* and *WinEndPaint()* functions to obtain the best possible performance from the system and to avoid output problems.

## The Micro PS

This kind of presentation space is also called *standard micro PS*, in order to distinguish it from the *cached micro PS*. An application cannot request a presentation space handle from the system, as was possible for the cached micro PS. Instead, the presentation space needs to be built directly in the program's code by calling *GpiCreatePS()*. This function returns a presentation space once it has been given a handle to a device context, the units of measurements of the presentation page, and one or more options referring to the creation mode selected.

```

#define INCL_GPICONTROL
HPS APIENTRY GpiCreatePS( HAB hab,
                        HDC hdc,
                        PSIZEL pszlSize,
                        ULONG flOptions ) ;

```

<i>Parameter</i>	<i>Description</i>
hab	Handle to the anchor block
hdc	Handle to a device context
pszlSize	Address of a SIZEL structure containing the dimensions of the presentation page
flOptions	Flags defining the attributes of the presentation space
<i>Return Value</i>	<i>Description</i>
HPS	Handle to a presentation space

The main difference with respect to a cached micro PS is the presence of a handle to a device context previously obtained through a call to *DevOpenDC()*. The presentation space must be associated with a device context; this establishes a link between the two elements, even before the output operations are started. Not all of the *GPI* functions can be exploited by using a micro PS; to access all of the graphics services present in PM it is necessary to use a normal presentation space.

## The Normal PS

A normal PS is the only presentation space of PM that allows the implementation of the schema depicted in Figure 3.2. To create a normal PS, you must use the *GpiCreatePS()* function, however, you don't have to indicate the handle of a device context. A normal presentation space can work without a device context. Once you need to display on any device the output image produced in the presentation space, you can establish the association between the two elements by calling the function *GpiAssociate()*:

```
#define INCL_GPICONTROL
BOOL APIENTRY GpiAssociate( HPS hps, HDC hdc) ;
```

<i>Parameter</i>	<i>Description</i>
hps	Handle to a presentation space
hdc	Handle to a device context
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The high degree of functional flexibility of a normal PS naturally involves a greater consumption of memory space with respect to the other two types of presentation spaces. When you use standard micro or normal presentation spaces, you need to specify a handle previously returned by the *GpiCreatePS()* function, as the second parameter of *WinBeginPaint()* in the code fragment processing the *WM\_PAINT* message. Thus, a *WM\_PAINT* message code looks like this:

```
...
case WM_PAINT:
{
    WinBeginPaint( hwnd, hps, NULL) ;
    ...
    WinEndPaint( hps) ;
}
...
```

The handle to the presentation space is not returned by *WinBeginPaint()*, but it has been created in another portion of the window procedure (likely during the interception of the *WM\_CREATE* message).



## The WM\_PAINT Message

All the logic governing the application's output operations is collected around the WM\_PAINT message. Keeping in mind what we have seen so far, the first operation is to gain access to a presentation space. Whatever kind of presentation space you will be dealing with, you will always have to use *WinBeginPaint()* and *WinEndPaint()* inside the WM\_PAINT message:

```
#define WINWINDOWMGR
HPS APIENTRY WinBeginPaint( HWND hwnd, HPS hps, PRECTL prectl );
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window within which output operations are to be performed
hps	Handle of a presentation space
prectl	Address of a RECTL structure
<i>Return Value</i>	<i>Description</i>
HPS	Handle to a presentation space

By looking at the parameters of this function, you can see that a presentation space is actually a set of data associated with a particular window, since it is necessary to specify that window's handle as the function's first parameter.

The second parameter refers instead to a presentation space. If you intend to obtain an hps, then you need to assign a NULLHANDLE to this value. You'll get a cached micro PS. Instead, provide a handle to a PS when dealing with a standard micro PS or a normal PS.

The last parameter defines the surface that will be used by the application for its painting operations. The value of NULL as the third parameter indicates the application's purpose of redefining the output portion of the window that corresponds to the *update region* (described below). After having terminated the painting activities, you have to release the handle to the presentation space by calling *WinEndPaint()*:

```
#define INCL_WINWINDOWMGR
BOOL APIENTRY WinEndPaint( HPS hps );
```

<i>Parameter</i>	<i>Description</i>
hps	Handle to a presentation space
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

It is always mandatory that the presentation space used in response to a WM\_PAINT message be provided to the application through a call to *WinBeginPaint()*.

## The Update Region

The presentation space returned by *WinBeginPaint()*, although theoretically the same as the one provided by *WinGetPS()*, exhibits fundamental differences in its inner

workings for generating the output. It has already been stressed that an OS/2 application must be able to cope with very frequent changes to its own output surface (the client window) whenever the window is resized or overlapped by the pixels of some other window. Both of these two events generate a WM\_PAINT message in the application's message queue. The WM\_PAINT message is an unusual message, both for the role it plays in an applications logic, as well as for its own nature and behavior. Its presence in an application's message queue implies the need to refresh some portion of the client window, an operation that can be very complex and computer intensive. To understand this statement, notice how the mouse's cursor disappears from the screen if it happens to be over a client window engaged in processing the WM\_PAINT message (it is the WinBeginPaint() function that acts on the mouse's cursor).

The presence of multiple simultaneous windows on the screen and the ease with which the user changes the size of any of those windows implies that the WM\_PAINT message is an event that happens often in any OS/2 application. Consequently, a lot of CPU time might be spent in performing painting operations which are often time consuming and complex.

On the basis of these considerations, let's examine the nature of the WM\_PAINT message. First of all, the presence of a WM\_PAINT message in an application's message queue implies the need to refresh part of the client window. The rectangle of the client window that is in need of refreshment is called the update region to indicate that the underlying pixels need to be updated. The update region is handled internally by the system and corresponds to the smallest rectangle that can possibly encompass all pixels that have been invalidated for some reason by the system or the application itself. The update region can grow in size very quickly (as more pixels become invalidated), for the very reason that repaint operations can be requested very often in PM. This means that the message queue of an application will often contain several WM\_PAINT messages that would all involve many update regions (that might even overlap each other). These considerations, together with the need to minimize the CPU overhead in repainting operations, have inspired the designers of the system to give special characteristics to the WM\_PAINT message.

The presence of invalidated pixels will not automatically generate a WM\_PAINT message unless the application's message queue is empty. Moreover, overlapping areas of invalidated pixels are consolidated into one larger, cumulative region to update.

Following this criterion, each OS/2 application will avoid immediate updating any invalidated region. There is also another reason for this. The flow of messages already present in the queue could bring about yet more changes that partially or totally invalidate what has already been updated immediately after receiving the first few WM\_PAINT messages. For this reason, a WM\_PAINT message is sometimes considered as a "low priority" message. It might be more helpful to consider WM\_PAINT like a message that shows itself in the queue only if there is an area of pixels that needs to be updated *and* the message queue is empty (Figure 3.3).

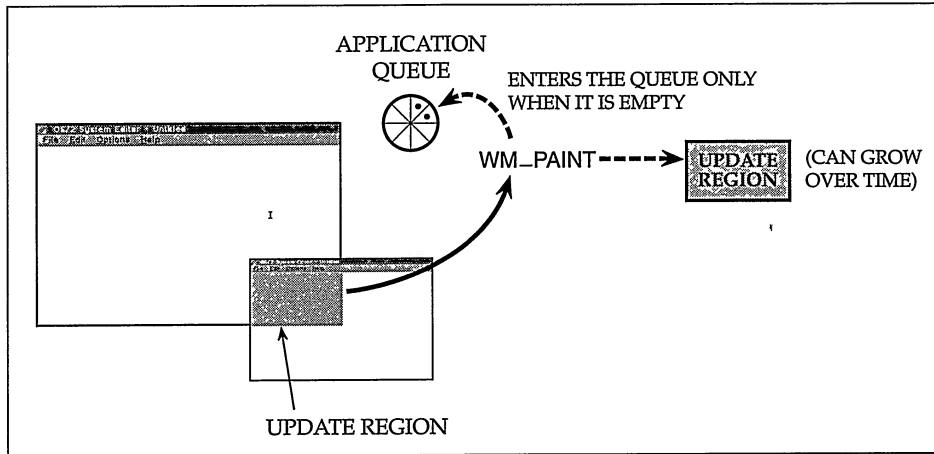


Figure 3.3 Generation of a WM\_PAINT message in the message queue of an application.

### Forcing a WM\_PAINT Message

The concentration of all output activities around one WM\_PAINT message also implies a development model that is based on different criteria from a simple application with a character-based user interface. In this case, whenever you need to display text on the screen, simply insert a *printf()* statement in the code, being careful not to overwrite portions of the screen that are already in use. In an OS/2 application such an approach is impossible, due to the strict rules that govern output, concentrating it around the WM\_PAINT message. So how can you possibly force an application to perform its output activity even when there is no WM\_PAINT message showing up in the queue? You simply simulate the sending of this message in such a way that the desired output statements, which are coded in the WM\_PAINT handling code, get executed and thus produce the desired results on the screen.

The tool that is most often used to achieve this is the function *WinInvalidateRect()*:

```
#define INCL_WINMESSAGEMGR
BOOL APIENTRY WinInvalidateRect( HWND hwnd,
                                PRECTL prectl,
                                BOOL fIncludeChildren );
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window to invalidate
prectl	Address of a RECTL structure corresponding to the rectangle that needs to be invalidated
fIncludeChildren	Boolean that indicates whether all child windows need to be included in the operation
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The window handle refers to the client, while the address of a RECT structure allows the identification of the rectangle that you want to invalidate. The boolean parameter indicates if the process of invalidating should involve all child windows of the window identified by the first parameter of the function.

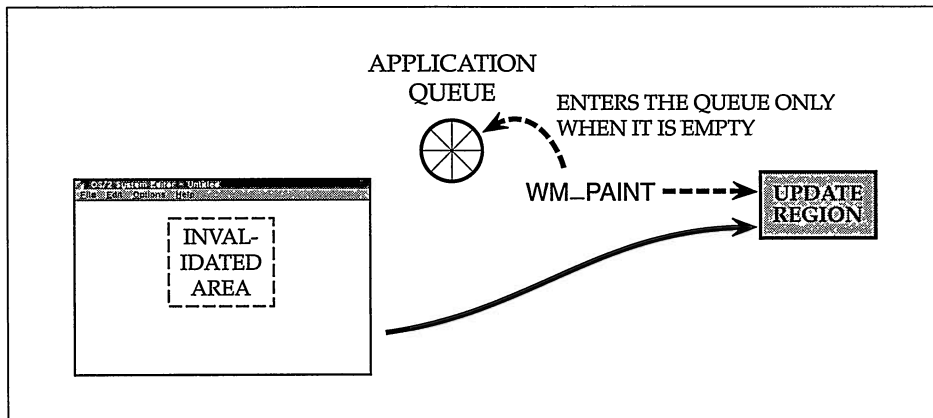
The second parameter is often replaced by a NULL value, meaning that the entire client has to be invalidated. The action performed by *WinInvalidateRect()* is creating an update region as large as that indicated by the second parameter, and then posting a WM\_PAINT message in the application's message queue. This message will eventually be retrieved by the message loop and passed to the appropriate window procedure, ensuring that the desired action occurs. Figure 3.4 summarizes the logic necessary to invalidate an area of pixels with *WinInvalidateRect()*.

PM's API also has the *WinUpdateWindow()* function, which refers to the handling of generating the output of the entire client window:

```
#define INCL_WINMESSAGEMGR
BOOL APIENTRY WinUpdateWindow( HWND hwnd ) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window within which all output operations are to be performed
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

This function sends a WM\_PAINT message directly to the appropriate window procedure, bypassing the message queue. However, this is not an alternative to *WinInvalidateRect()*. In fact, *WinUpdateWindow()* will perform its task only if there already exists an invalidated area for the window indicated by the handle and for any possible child windows that might be present. If there is no update region, the function



**Figure 3.4** Contrivance for forcing a repaint of the whole or part of the client window.

has no effect and does not issue the WM\_PAINT message. Very often, it is convenient to combine *WinInvalidateRect()* and *WinUpdateWindow()*.

```
...
WinInvalidateRect( hwnd, NULL, TRUE ) ;
WinUpdateWindow( hwnd ) ;
...
```

With the first function you define the invalidated region by inserting a WM\_PAINT message into the message queue. The subsequent call to *WinUpdateWindow()* retrieves the WM\_PAINT message from the queue and delivers it to the appropriate window procedure, thus updating the invalidated area. Using this method, you can be sure that the repainting of the client window will take place almost instantaneously, since the message handled by *WinUpdateWindow()* will immediately reach its processing code.

## Output Synchronization

The combined effect of a *WinInvalidateRect()* followed by a *WinUpdateWindow()* allows you to override the normal, asynchronous painting activities performed by PM. There is another means by which to make the painting activities synchronous that does not require you to resort to the combination of *WinInvalidateRect()* and *WinUpdateWindow()*. This solution can also be extended to all those conditions not directly controlled by the application code (a window's overlapping a portion of a client window or resizing).

Among the registration styles of a class of windows there is also CS\_SYNCPAINT. In the API we find also the analogous WS\_SYNCPAINT used when a window is created. The presence of this style (at the class level or at the single window level) modifies the behavior of *WinInvalidateRect()*, so that it sends a WM\_PAINT message directly to the appropriate window procedure rather than "parking" it near the application's message queue. In this case, all painting operations will be terminated before the function returns, thus making WM\_PAINT behave like any other PM message. In other words, WM\_PAINT will not be penalized in the message queue. (Actually a WM\_PAINT message will not even appear in the message queue of an application that employs windows belonging to a class registered with the CS\_SYNCPAINT style or windows created with the WS\_SYNCPAINT style.)

The software designer is thus free to choose between implementing the output operations with a successive consolidated update region and placing a WM\_PAINT message in the message queue, provided the queue is empty (asynchronous painting), or with a direct immediate processing that successively deals with all update regions (synchronous painting) in sequence.

Both solutions are valid. The choice depends on how "heavy" and complex the output task is. If you are sure that the size of the update region is limited, and that it will seldom happen that any nearby pixels might be invalidated, then it might be worthwhile to adopt a synchronous painting technique. On the other hand, if the application changes the client window's pixels often and generates complex images,

then it is better to follow PM's standard approach, asynchronous painting. Figure 3.5 summarizes the different behaviors of *WinInvalidateRect()* and *WinUpdateWindow()* according to the alternative painting techniques employed.

## Output Handling

PM optimizes the presence of the `WM_PAINT` message in order to minimize any ineffective repainting operations that could happen due to the high frequency of interactions of the multitasking environment. The concentration of all painting instructions inside the block enclosed by the *WinBeginPaint()* and *WinEndPaint()* calls in the `WM_PAINT` means that all output statements have to be contained between these two limits. The application can control, through some flags, whether certain portions of these output statements are executed or not. For example, imagine you're writing a program to draw some circles with a *DrawCircle()* function you have designed. *DrawCircle()* is called only if the user has selected an appropriate menu option in the program. The following code fragment reproduces this hypothetical situation:

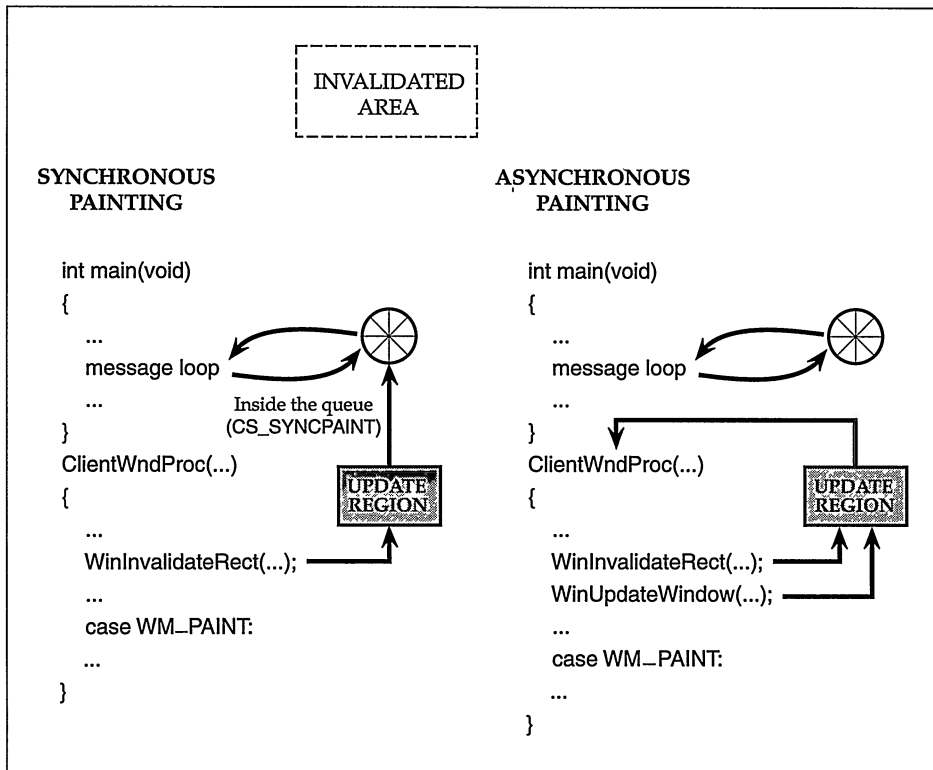


Figure 3.5 *WinInvalidateRect()* and *WinUpdateWindow()*.

```

...
case WM_PAINT:
{
    HPS hps ;

    hps = WinBeginPaint( hwnd, NULLHANDLE, NULL) ;
    DrawXMasTree( x, y, sHeight) ;
    if( fBalls)
        DrawCircle( x1, y1, sDiameter, clrColor) ;
    ...
    WinEndPaint( hps) ;
}
break ;
...

```

The *DrawXMasTree()* function displays a Christmas tree with no ornaments. These will appear only if the boolean identifier *fBalls* is nonzero. In our hypothetical program, *fBalls* takes on the value of *TRUE* only if the user makes a determined selection, which will cause the client window to become invalid. Eventually the window procedure will receive a *WM\_PAINT* message and will take care of drawing the Christmas tree with all ornaments in place.

This type of control actions to be performed within the governing logic of the *WM\_PAINT* message—that is, when the application is about to generate its output—relies completely on how the application has been coded. However, there is also another criterion for determining which statements should be executed when processing the *WM\_PAINT* message. Despite the fact that the entirety of the code for displaying text and images is contained in this portion of the window procedure, the application does not have to execute *all* of these statements.

This is not a programmer-defined criterion, rather it is something governed by the value and the size of the update region. PM is clever enough to implement a selective execution of all basic statements that make up the block of code contained between *WinBeginPaint()* and *WinEndPaint()*. To fully understand what this means, remember that all of the *Gpi* and the *Win* functions that deal with output generation—for instance, *WinFillRect()*—require as their first parameter a handle to a presentation space. The *hps* used during processing of the *WM\_PAINT* message is a handle that is “update-region aware.” This means that amid all information contained in a presentation space—information that might not be directly accessible to the application—there are some pieces of data identifying the update region. When PM needs to execute a *Gpi* or a *Win* function that requires an *hps* as its first parameter, its internal logic will test whether the output produced overlaps the area invalidated by the system.

Thus, it can often happen that, although present in the code processing the *WM\_PAINT* message, one or more functions will not be executed, because their output would not affect the invalidated area. Were they executed, the result would be exactly the same as what is already displayed in the window. We will test this in Listing 3.7.

Let’s apply this concept to the Christmas tree example. You can now imagine that the appearance of decorations will concern only limited portions of the entire drawing,

and therefore it would be ineffectual to redraw the entire picture. The function *DrawXMasTree()* will not be executed by PM for each addition made to the drawing. It will be called only when some pixels covered by the Christmas tree's image happen to be in the invalidated area. This action, which is completely transparent to the application, allows the reduction to a bare minimum of any intervention on the part of the system for performing all painting operations, and thus avoids overloading the processor. Figure 3.6 summarizes the behavior of PM as far as the handling of the WM\_PAINT message is concerned.

The behavior of PM when generating output in a window's client area clarifies unequivocally why *WinBeginPaint()* rather than *WinGetPS()* is used to obtain a presentation space handle. This second function returns a presentation space handle that is not "update-region aware," and therefore would not allow for an appropriate and optimal execution logic of output generation. This means that the presentation space handle obtained through *WinBeginPaint()* must be passed back to the system by means of *WinEndPaint()* only within the block of code handling the WM\_PAINT message.

## Erasing a Window's Background

Now we know about all elements needed to paint an application's client window. The changes to make to the code are concentrated only in the window procedure. The first

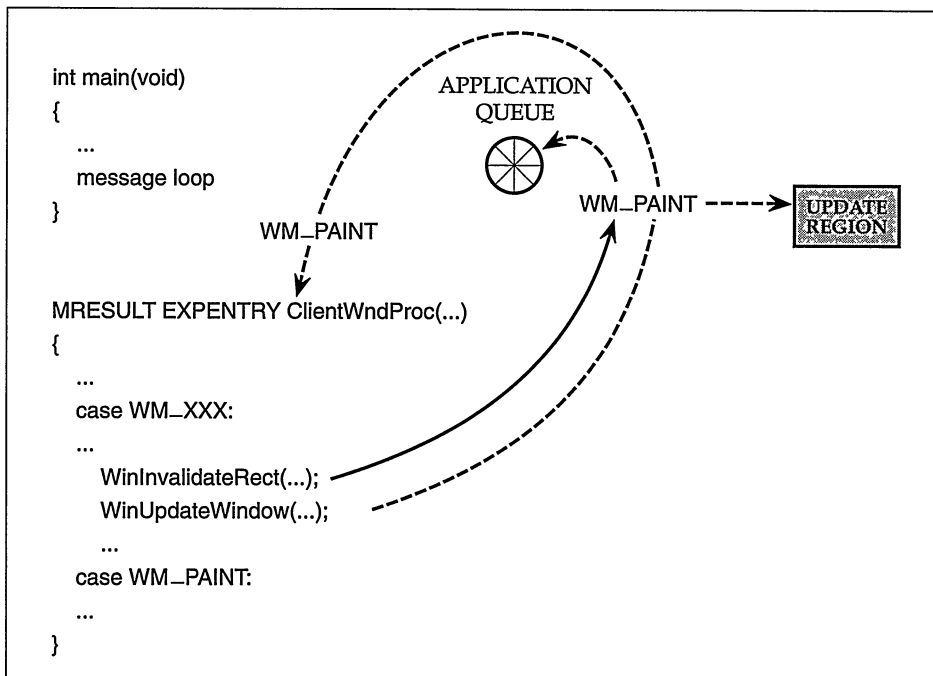


Figure 3.6 Logical scheme governing the painting mechanism in PM applications.



solution (Listing 3.2) presented, is also the more complicated, and is based on the usage of *WinFillRect()*; therefore it is necessary to declare an identifier of type HPS and another one of type RECTL.

**Listing 3.2 The Processing of the WM\_PAINT Message Now Permits You to Color in White the Application's Client Window**

---

```
MRESULT EXPENTRY ClientWndProc(  HWND hwnd,
                                ULONG msg,
                                MPARAM mp1,
                                MPARAM mp2)
{
    switch( msg)
    {
        case WM_PAINT:
        {
            HPS hps ;
            RECTL rc ;

            hps = WinBeginPaint( hwnd, NULLHANDLE, &rc) ;
            WinFillRect( hps, &rc, CLR_WHITE) ;
            WinEndPaint( hps) ;
        }
        break ;

        default:
            break ;
    }
    return WinDefWindowProc( hwnd, msg, mp1, mp2) ;
}
```

The presentation space handle returned by *WinBeginPaint()* is aware of the invalidated area that is stored in the RECTL structure specified as the function's third parameter. This change, in comparison to the *WinBeginPaint()*, makes the rectangle that needs to be filled with the function *WinFillRect()* immediately accessible. The following list shows all the defines for colors that are usable with *WinFillRect()* and other PM API functions.

<i>Define</i>	<i>Value</i>
CLR_WHITE	(-2L)
CLR_BLACK	(-1L)
CLR_BACKGROUND	0L
CLR_BLUE	1L
CLR_RED	2L
CLR_PINK	3L
CLR_GREEN	4L
CLR_CYAN	5L
CLR_YELLOW	6L

CLR_NEUTRAL	7L
CLR_DARKGRAY	8L
CLR_DARKBLUE	9L
CLR_DARKRED	10L
CLR_DARKPINK	11L
CLR_DARKGREEN	12L
CLR_DARKCYAN	13L
CLR_BROWN	14L
CLR_PALEGRAY	15L

In Figure 3.7 you can see the final result produced on the screen by the ERASE applications.



Listing 3.3, instead, presents the alternate solution, given by the usage of the *GpiErase()* function. In this case, it is necessary to insert the `#define INCL_GPICONTROL` directive in the first few lines of the application in order to have readily accessible the function's prototype.

In the case of *GpiErase()*, the selected color corresponds to the system default, `CLR_BACKGROUND`. The solution presented in Listing 3.3 is one of the most common in OS/2 applications. With this second approach it is not necessary to declare a `RECTL` structure as in the preceding example.

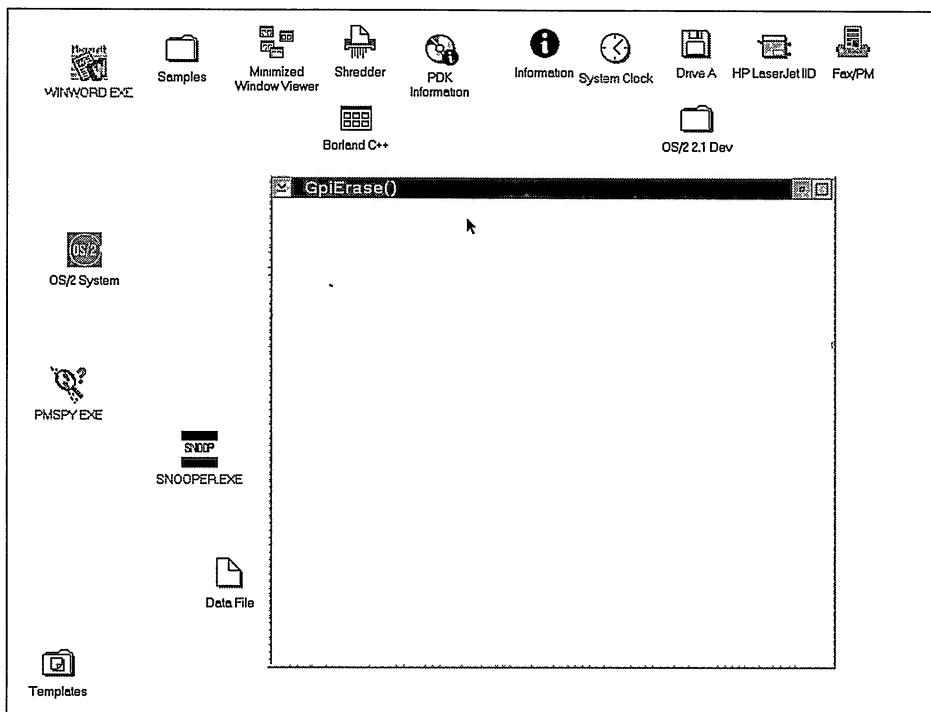


Figure 3.7 ERASE with its client window colored in white.

## Message Flow in PM

The message `WM_PAINT` has allowed us to solve a practical problem with the sample `MACHINE` application. However, we have also noted that it features some unusual characteristics not found in most of the `WM_` messages of PM's API.

### Queued Messages

Before examining other messages, it is necessary to discuss a distinctive element represented by the different arrival modes of a message in a window procedure. Up to this point, we have seen that the final receiver of a message is the window procedure of the class to which the target window belongs. (Remember that an application can have more than one class.) More precisely, the presence of a message in PM is always *addressed* to a window, that is, to its handle. The processing of the message takes place in the window procedure of the class to which the window belongs.

If a window procedure is in all cases the final addressee of all messages, they can reach the window via two different paths: by means of the application's message queue, or directly skipping the message queue altogether. Figure 3.8 represents the first method.

By this method, a message passes from the system's message queue to the message queue of the application, where it is read and retrieved by the message loop and finally passed to the window procedure. This is called *posting* a message, sometimes called *queued messages* or *asynchronous messages*.

The first definition corresponds to the idea of physically posting a letter in a mail box (a message in the system queue), and to deliver it through the addressee's local post office (the application's message queue). The address of the addressee is represented by the handle of the window to which the message refers; in almost all cases, though, the sender of the message is unknown.

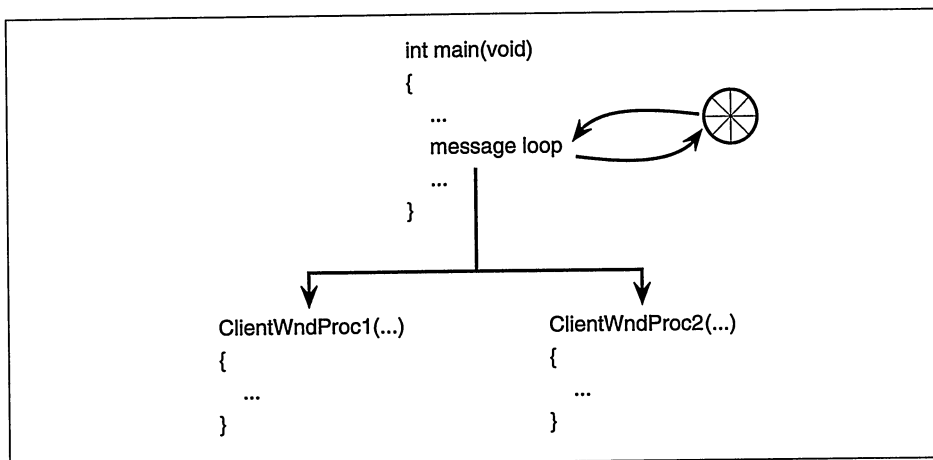


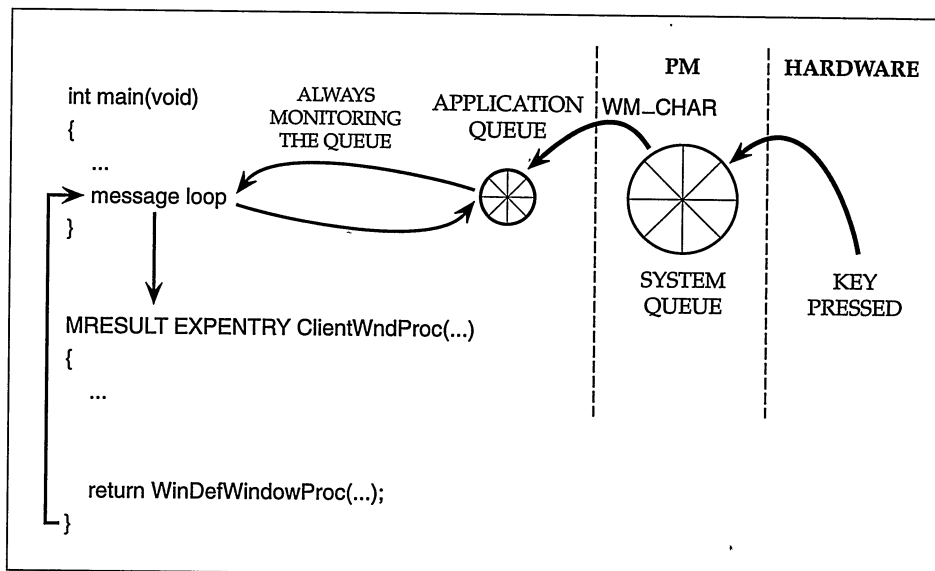
Figure 3.8 Flow chart of queued messages reaching a window procedure through the application queue.

This type of message is also known as an *asynchronous* message, because its processing in the window procedure of the addressee window will happen at some later moment when the message is posted into the queue. There is an interval—which is difficult to measure due to the high degree of variability of the environment—between the moment when a message is inserted into and the moment it is recovered from the application's message queue.

Figure 3.9 illustrates this last point. Imagine that the user presses the A key on the keyboard. The physical action is interpreted by the keyboard driver and it is then passed on to PM. PM will in turn translate the data it receives into the WM\_CHAR message, and then takes care of posting that message in the application that was active in PM at that very moment. The function *WinGetMsg()* will retrieve the message and transfer it by means of *WinDispatchMsg()* to the window procedure of the class to which the addressee window belongs. The message will wait for a variable amount of time inside the window procedure, according to the kind of processing that it will be subject to. It can be just a fraction of a second, or a significant amount of time.

Naturally, you will try to develop applications that minimize the interval between a message's entrance into and exit from the queue. To avoid situations with longer waiting times of the various messages in their respective queues, it is necessary that all PM applications stick to some structural and behavioral rules, so that they don't tie up the CPU for longer periods of time. We will examine these techniques in Chapter 9.

Before going into more detail, you need to know more about the addressee of a posted message. It will invariably be a window belonging to the application. An OS/2 application



**Figure 3.9** Message processing scheme for a message retrieved from an application's message queue.

usually registers different classes, each of which can be represented by several windows. Here are the possible scenarios:

- Message passing between two windows of the same class
- Message passing between two windows of different classes
- Message passing where both the sender and the receiver are the same window

The *WinPostMsg()* function is the tool you will use to post a message:

```
#define INCL_WINMESSAGEGR
BOOL APIENTRY WinPostMsg( HWND hwnd,
                          ULONG msg,
                          MPARAM mp1,
                          MPARAM mp2) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the message's addressee window
msg	Numeric value of the message
mp1	Additional information
mp2	Additional information
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

You may have noticed how the parameters to this function are exactly the same as those required by a generic window procedure. This should not come as a surprise, since a window procedure is in any case the final target of any message. The usage of *WinPostMsg()* requires you to know the handle of the addressee window. This piece of information is obviously available when the receiver is the same as the sender. However, it should not be difficult to know the sender handle even when two different windows are involved, simply because they will be related one to another in some way.

The solution given by *WinPostMsg()* cannot be exploited directly by PM for implementing a communication means between windows belonging to two different applications. For this type of problem, you need to resort to the communication protocol known as DDE (explained in Chapter 11).

The *WinPostMsg()* function returns a boolean value that indicates the success or failure of the insertion operation of the message into the queue. It is good programming practice to always test this return value in order to make sure the operation was successful. In general, the size of a PM application's message queue will not be a limiting factor for the program's execution. The ten messages that a standard message queue can accommodate are more than adequate for most applications. Only those applications that make heavy usage of the DDE communication protocol might need a larger queue.

## The Parameters of `WinPostMsg()`

The four parameters of the `WinPostMsg()` function must include the appropriate data: the handle of the target window, the type of message, and possibly some additional information.

Any of the messages listed in Table 3.1 can be posted with `WinPostMsg()`. In general, `WM_` messages are those used most frequently to respond to most situations. In Chapter 7, we will examine the predefined windows classes and see what specific message can be used with the various kinds of windows.

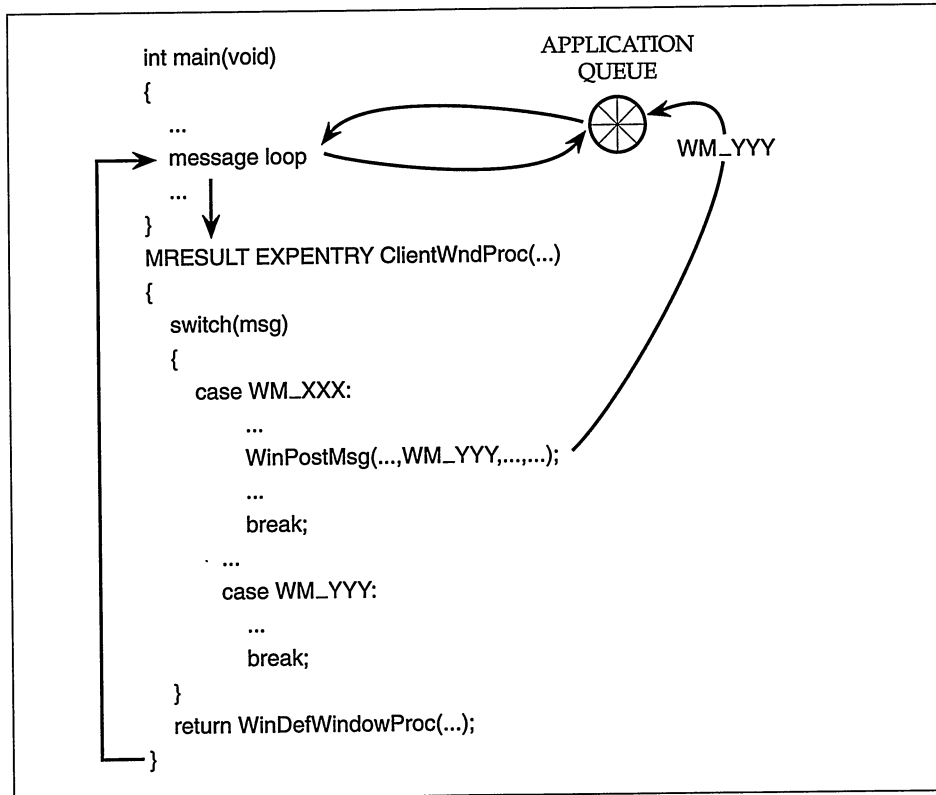
Each message contains some additional information that is inserted in the two 32-bit `MPARAM` identifiers. The Toolkit documentation will indicate what kind of information should be inserted into `mp1` and `mp2`. In some cases, one of the two parameters is reserved or does not convey any special meaning for the message's inner working.

## When to Post a Message

The messages explicitly posted into the application queue by means of `WinPostMsg()` are asynchronous. Why should you post a message? Aside from the final receiver, posting a message into the application's queue follows the guideline of executing a particular portion of code of a window procedure—precisely that portion of code of the class to which the destination windows belongs.

A window procedure is a function that is characterized by having its code divided in several smaller “blocks” corresponding to the various messages that need to be caught. If, during the processing of a message you need to call `WinPostMsg()`, then it means that the application's logic will have to execute the code fragment associated with the posted message. But when is this to happen? Later, at some moment after the current message's processing. Posting a message is like scribbling a note and attaching it to someone's PC screen. You wish to communicate something, and maybe even ask that something be done (like making a phone call) within a reasonable amount of time—certainly later than when you actually stick the note to the screen. Let's look at Figure 3.10.

A generic window procedure has received a `WM_XXX` message that has been intercepted correctly. The logic behind its processing implies various operations (the ellipses) and a call to `WinPostMsg()` to post the `WM_YYY` message into the queue. Once the operation is complete, `WM_XXX`'s processing goes on and eventually terminates with the `break` keyword that yields to `WinDefWindowProc()`. Exiting from the window procedure happens with the return value of `WinDefWindowProc()`, and reenters the message loop in `main()`. Here, after `WinDispatchMsg()` returns, another iteration calling `WinGetMsg()` is executed. We don't know which and how many messages are present in the queue. However, it is certain that the `WM_YYY` message is there, simply because it has been inserted there by the call to `WinPostMsg()`. Let's assume that `WM_YYY` is preceded in the queue by two other messages. After those two messages have been processed, it will be `WM_YYY`'s turn. The message reaches the window procedure of the class to which the target window belongs (in the example, the same window procedure in which `WM_XXX` was previously intercepted).



**Figure 3.10** Posting a message.

What we have just described corresponds, in broad terms, to most of the operations needed to terminate an application. In this case, the message to be posted is `WM_QUIT` (corresponding to `WM_YYY` in the above description). The only difference to be found in the invalidation of the message loop when `WM_QUIT` is recovered is in its not being passed to a window procedure.

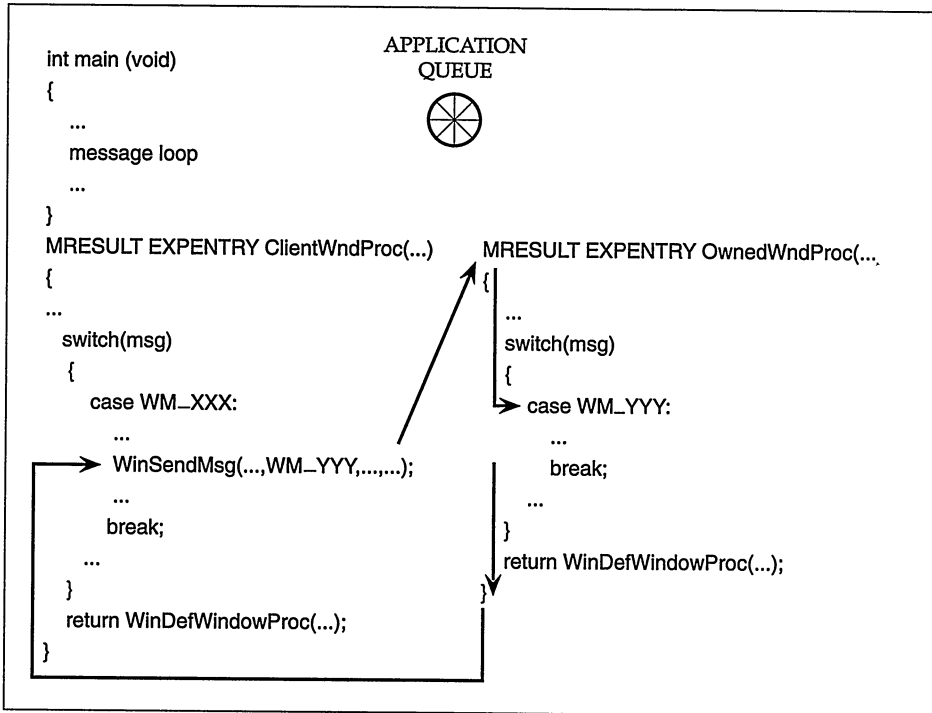
## Non-Queued Messages

The second solution for sending messages is by reaching the target window procedure directly, skipping the application's message queue altogether, and therefore bypassing the application's message loop. The tool used for this purpose is the `WinSendMessage()` function of PM's API, a function that takes on the same syntax as `WinPostMsg()`:

```

#define INCL_WINMESSAGEGR
MRESULT APIENTRY WinSendMessage(  HWND hwnd,
                                  ULONG msg,
                                  MPARAM mp1,
                                  MPARAM mp2) ;

```



**Figure 3.11** Flow chart of a non-queued message sent via `WinSendMessage()`.

<i>Parameter</i>	<i>Description</i>
<code>hwnd</code>	Handle of the message's target window
<code>msg</code>	Numerical value of the message
<code>mp1</code>	Additional information
<code>mp2</code>	Additional information
<i>Return Value</i>	<i>Description</i>
<code>MRESULT</code>	Return value of the window procedure that processed the message

The distinctive feature of `WinSendMessage()` is that it will send the message specified in `msg` directly to a window procedure. Therefore it is correct to speak about synchronous processing, because this function does not return any value until the target window procedure has terminated all operations triggered by the message. Figure 3.11 summarizes the situation.

The reception of a `MRESULT` return value corresponds to the return type of a generic window procedure. Also `WinSendMessage()` is to be used like a communication tool within one application. But what is the target window of a message being sent this way? Can the receiver and the target window be the same? In this case there are several possible scenarios:



- Sending and receiving windows belong to different classes
- Sending and receiving windows belong to the same class
- Sending and receiving window are the same

Just as for *WinPostMsg()* and *WinSendMsg()*, a window can send a message to itself, or to another window of the same or another class. The fundamental difference between *WinSendMsg()* and *WinPostMsg()* is in the certainty that the action triggered by the sent message will be performed by the application, and that this will happen as soon as the message has been sent. In general, this function is used more frequently than *WinPostMsg()*.

## When to Send a Message

The following discussion will help you to better understand how to take advantage of message passing via *WinSendMsg()*. In several OS/2 applications, especially before the advent of WPS, there were dialog windows to access the file system and to load a file into memory. The choice of a file takes place in a list box, and the confirmation of the user selection is done by pressing the OK button. You can also double-click on a filename without pressing the OK button. In practice, you can achieve the same result via two different actions. A double-click or the pressing of the OK button are two distinct events, and they are processed in two different code fragments within the applications. However, the actions performed in those two code fragments are very similar, but not the same. When you double-click, the application must retrieve the file, open it, and load it. When you press the OK button, all these actions must be preceded by a test that a filename is selected. A very common solution to this problem is to perform the test in the code fragment dealing with the OK button, and then the program simply sends itself a message that simulates what would happen in the system when the user double-clicks. In this way, the program can grant that the code programmed for the double-click—the loading of the file—is executed. Figure 3.12 illustrates this process.

## Some Message-Sending Considerations

At this point, the whole picture might seem somewhat complicated, especially if you are new to PM. When should a message be sent and when should it be posted? What criteria should govern the first mechanism, and what the second?

There is no simple rule that can determine whether some kind of message should be posted or sent; only experience and careful study of sample listings will give you a set of rules that can help you. The solution given by *WinSendMsg()* is the most common, because it allows one or more actions to take place immediately. The result is similar, in a certain sense, to making a direct call to a code fragment of the same or of a different window procedure, as if it were a function or a jump (like the GOSUB-RETURN paradigm of BASIC). Each code fragment corresponding to an intercepted message in a window procedure can be thought of as a separate function that can be called by sending the corresponding message with *WinSendMsg()*.

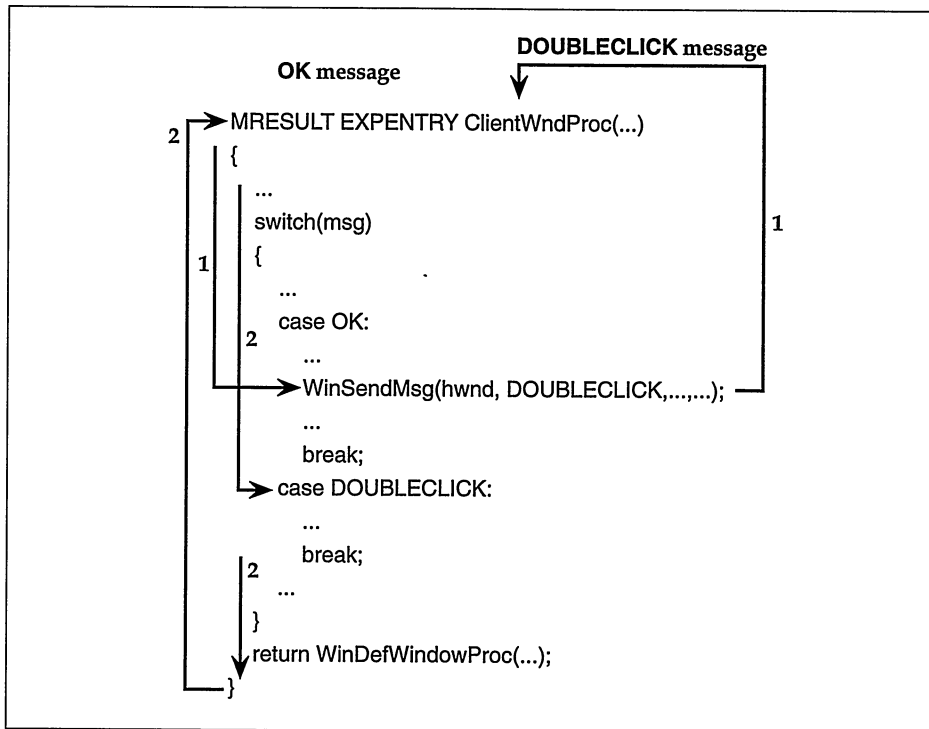


Figure 3.12 Scheme of message passing to minimize writing new code.

## Functions and Messages

The MACHINE application developed in Chapter 2 is limited to displaying on the screen a window characterized by a few structural elements, like the titlebar, icons, and the sizing border. This isn't much compared to industrial strength programs which are available. However, the few lines that make up MACHINE do indeed generate a great many messages that are all sent to the only window procedure present in the code. The message flow, at least for now, is completely hidden to the application, due to the action of `WinDefWindowProc()` that collects all messages. Your best shot at discovering what is happening inside the application when you create a window is by using the IPMD debugger. For this purpose MACHINE must have been compiled and linked with appropriate switches. From `WorkFrame/2` you can then use the accelerator `Ctrl+D` to access the IPMD's interface windows. The best point of view for seeing the message going to a window procedure can be established by setting a breakpoint exactly on the statement:

```

...
switch( msg)
...

```

To set a breakpoint with IPMD, just double-click the left mouse button on the line number of the source code under examination, as in Figure 3.13.

To evaluate what goes on inside the program, you execute the code step-by-step, with the trace functionality activated by pressing the right mouse button or through hot keys like Ctrl+O or Ctrl+I. The execution of the code proceeds statement by statement, showing on-screen what is actually happening in the PM screen group in reaction to the execution of every code statement.

The function *WinCreateStdWindow()* is an important step in the logic flow of our application because it corresponds to the first appearance on the screen of the program's main window. Once you press the right mouse button, when the horizontal cursor bar is positioned on the first parameter of *WinCreateStdWindow()*, you will find yourself directly inside the class's window procedure.

This jump from *main()* to *ClientWndProc()* was not caused by a direct call, or by the passing of any explicit message (as there is no *WinSendMsg()* call in sight). The jump in the execution flow into the window procedure is due to *WinCreateStdWindow()*.

The debugging performed up to this point suggests that it is the *WinCreateStdWindow()* function that produces as a side effect the passing of one or more messages directly to the window procedure of the class to which the window belongs. This means that the messages can arrive at a window procedure not only on behalf of the programmer's choice, as the programmer calls functions like *WinSendMsg()* and

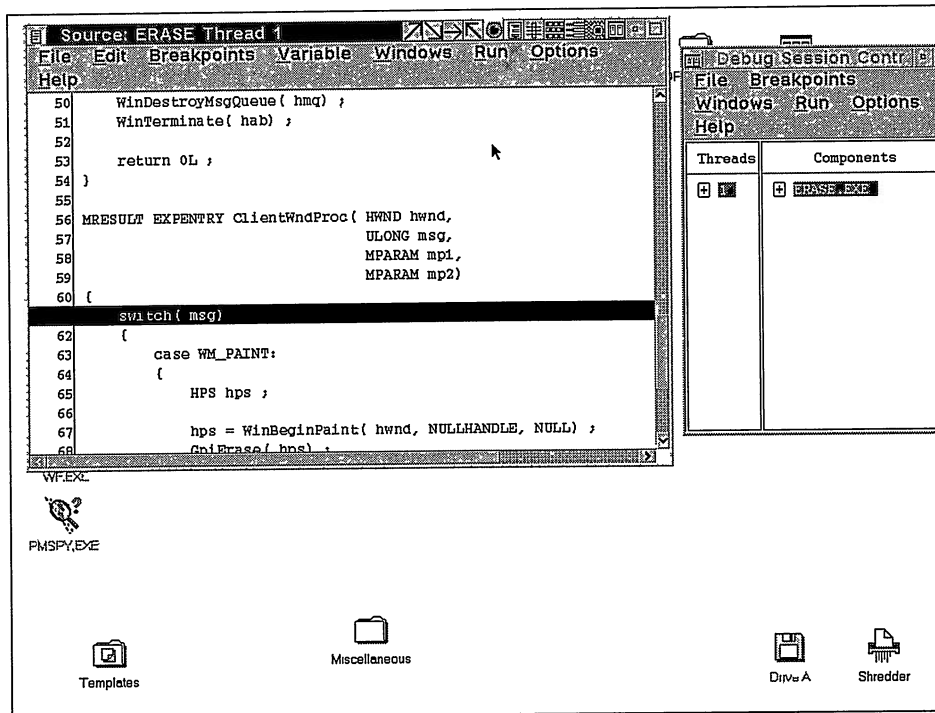


Figure 3.13 IPMD with a breakpoint set right in the window procedure.

*WinPostMsg()*, but also as a direct consequence of using several of PM's API functions. This point is critical to PM programming.

In general, within the category of non-queued messages, there are two cases:

- Messages explicitly sent by means of the *WinSendMessage()* function
- Messages generated as a side effect of the usage of some of PM's API functions

The application's flow of execution moves into the window procedure specified at class registration time in a call to *WinRegisterClass()*. The parameters given to the window procedure are the same as the first four members of a QMSG structure, but they do not originate there, since at this moment no read operation from the message queue has been performed. The type of message sent directly from an API to a window procedure of the window's own class as a consequence of a side effect can vary from function to function and are part of the side effect chain itself.

This consideration becomes a fundamental piece in the development model of PM. It is necessary that the programmer know the set of side effects that are produced by PM's API function calls in order to take advantage of them. However, very often, the messages generated by the API functions are not described in the technical documentation. To get hold of these invaluable facts, the best method is to explore step by step the evolution of a running application and retain the series of messages received by the window procedure.

To determine which message actually arrives in the window procedure, you can open IPMD's Program Monitor List window and ask it to keep track of the msg identifier. The most convenient representation is the hexadecimal format, the same used in PMWIN.H to describe every single message (see Appendix C for a quick reference to these numeric codes).

As you proceed with the debugging phase, you will see that with *WinCreateStdWindow()* only two messages are sent to the window procedure. The reception of the first message of the flow causes *WinDefWindowProc()* to generate other related messages. This behavior is not fixed—it depends on the flags set when the class was registered or the window was created. Table 3.2 summarizes what has emerged from the analysis through IPMD of the code reproduced in Listing 3.4, without changing the structure of the module definition file (Listing 2.8) or the makefile (Listing 2.7).

Even before returning the value of the window's handle, the *WinCreateStdWindow()* function sends the WM\_CREATE message to its own window procedure. In response to that, *WinDefWindowProc()* sends a WM\_ADJUSTWINDOWPOS message, as shown in Table 3.2.

**Table 3.2** The *WinCreateStdWindow()* Function, as defined in Listing 3.4, generates this pair of messages sent directly to the window procedure.

<i>Numeric Value</i>	<i>Message</i>
0x001	WM_CREATE
0x008	WM_ADJUSTWINDOWPOS

Eventually, the flow of execution returns to the *main()* function, where it reaches the execution line of *WinShowWindow()*. This function, in turn, will send the WM\_FORMATFRAME to the window procedure and generate the sequence shown in Table 3.3.

The sequence of messages listed in Tables 3.2 and 3.3 would have been unique and would continue omitting the *WinShowWindow()* function and setting the WS\_VISIBLE flag in *WinCreateStdWindow()*. Therefore it is not possible to establish a clear-cut criterion that allows you to determine what messages pertain to a particular API function, because the role played by the various style flags affects them a great deal. In many cases, the only tool by which to discover what is actually happening inside an application is IPMD.

## Messages, Windows, and Window Procedures



The creation of a window happens with *WinCreateStdWindow()*. This function returns a handle to the frame window and the handle to the client window appears in the last parameter. In the sample Listing 3.5, that means that the handle of the frame window is a known entity in *main()* once *WinCreateStdWindow()* has terminated its execution. However, due to side effects, some messages arrive at the class's window procedure before the function returns.

Remember that a message must be sent to a target window—even *WinCreateStdWindow()* must follow this rule. Which window? It can only be the application's window (i.e. the application's client window). A window's handle is first known in the window procedure of the class to which the window belongs, rather than in the function that calls *WinCreateStdWindow()*. Therefore, it is not necessary to declare a source file scope identifier for storing a window's handle, because this is always passed automatically by PM when a message is sent to the window procedure. Keep in mind:

- A message is always addressed to a window (there can be some variations like sending a message to an application queue)
- Messages are always processed in a window procedure
- In each application instance there will be a window procedure busy processing one message at a time, and it always knows to which window that message is addressed

**Table 3.3** Messages Generated by *WinShowWindow()* in Listing 3.5

<i>Numeric Value</i>	<i>Message</i>
0x0041	WM_FORMATFRAME
0x0055	WM_WINDOWPOSCHANGED
0x0007	WM_SIZE
0x0005	WM_SHOW
0x004F	WM_ERASEBACKGROUND
0x0023	WM_PAINT

## Functions That Use Messages

The analysis of the preceding output techniques allowed us to conclude that the *WinInvalidateRect()* function has as a default side effect the posting of a *WM\_PAINT* message in the application's queue. In addition to functions that send messages directly to the appropriate window procedure (like *WinCreateStdWindow()*, *WinCreateWindow()*, *WinUpdateWindow()*, *WinShowWindow()*, *WinSetWindowPos()* and others), there are other functions that have as a side effect the posting of a message in the target application's message queue. The flow of information in PM is therefore the direct consequence of at least four different events:

- Sending messages
- Posting messages
- Use of API functions that send messages
- Use of API functions that post messages

In special cases, some functions can at times post a message and at other times send it directly (that is what happens with *WinInvalidateRect()*, according to the painting technique chosen for the window). Despite the source, the addressee is a window and the final destination a window procedure.

## The Messages of the Predefined Classes

In Table 3.1 you can see a list of the message prefixes related to the predefined window classes. It's just a series of numerical defines that are different from generic *WM\_* messages. In Chapter 7 we will examine in greater detail the messages of each single class. Each of these messages follows the rules discussed in the previous paragraphs with a preference for the sending mechanism.

## Defining New Messages

In addition to the standard *WM\_* messages and those belonging to the predefined classes, there is also the possibility of defining new messages to satisfy particular application requirements. In *PMWIN.H* there is the message *WM\_USER* that takes on the value of *0x1000*, greater than that of any other message. *WM\_USER* is the starting point for defining new messages that the programmer can pass to windows belonging to the same program. The definition of a new message is based on a simple system that takes advantage of the C compiler's *#define* preprocessor directive. In several places in this book the *WM\_PASSPROC* will be defined in the following way:

```
#define WM_PASSPROC      WM_USER + 0
```

In general, this *define* will be placed in the application's header file or in the first few lines of source code. *WM\_PASSPROC* becomes a message that can be transferred via function calls like *WinPostMsg()* or *WinSendMsg()*. Naturally, the target window must be of a class that is capable of catching and processing the message(s) defined by the

application. The messages defined in the code will never be sent by some API function, not even as side effects.

Messages defined in your code follow the same standard rules. You send them through *WinSendMessage()* or you post them via *WinPostMessage()*.

## Some Enhancements

Intercepting the message flow directed to a window procedure by setting a breakpoint with IPMD is a very rewarding way to learn how a PM application works. The display of a message's hexadecimal value alone, however, is not completely self-explanatory. It would be much more convenient and useful if it were possible to show in IPMD's Program Monitor List window the name assigned to the message in PMWIN.H. This can be done by declaring a bidimensional array of characters containing the text string of each single message. However, it is not the best approach as far as C programming is concerned or for the development model in PM. A better solution is to define a string table inside the resource file, and then read the strings from within the code at execution time. In Chapter 5 we will see all the elements that will allow us to build such a solution.

## Message Parameters



In Listing 3.5 the flow of execution in the window procedure goes through the `WM_PAINT` case branch when the value of the `msg` identifier is equal to `0x0023`. Once that piece of case code has been executed, after abandoning the `switch` code block, execution returns to *WinDefWindowProc()* and then to *main()*. In the case of `WM_PAINT`, we face a very special situation, because this message is involved with all of the application's output activities. However, the rule extends generally to all messages. A window procedure will always receive at least once the `WM_CREATE` message, issued by the *WinCreateStdWindow()* function. We will now change the structure of the window procedure in Listing 3.4, in order to test for a `WM_CREATE` condition (Listing 3.5).

The `WM_CREATE` message contains some interesting information in the `mp1` and `mp2` parameters. This consideration is valid for almost all of the messages implemented in PM (some of them will have an empty parameter, but most of them will exploit all 32 bits available for describing the nature of the message involved).

The technical documentation of the Toolkit indicates that the `WM_CREATE` message submits in the `mp1` parameter a type `PVOID` parameter. The area to which it refers contains data that is specific to that window. When a `WM_CREATE` message is received, the program is engaged in executing the *WinCreateStdWindow()* function for generating a window. The `WM_CREATE` message is sent to the window procedure of the class to which that window belongs. Via `mp1` the programmer has a chance to access information critical for the window's creation. The syntax of *WinCreateStdWindow()*

does not allow the specification of any information to be retrieved by means of the `mp1` parameter of `WM_CREATE`. So, if we intercept the `WM_CREATE` sent by `WinCreateStdWindow()`, `mp1` will be `NULL`.

In the case of Listing 3.5 we have not dealt with any piece of information that is critical for the window's creation.

It is far more interesting to examine the value of `mp2` (Figure 3.14). This second parameter is a pointer to a `CREATESTRUCT` structure. By using IPMD to review the memory area pointed to by `mp2` you will discover some values related to the `WinCreateStdWindow()` function. In Chapter 4 we will discuss the contents of the `CREATESTRUCT` structure. It is evident that even in the window procedure you can find some data relating to a window's parent. The member `hwndParent` of the `CREATESTRUCT` structure takes on the same value that the `hwndFrame` identifier has in the `main()` function. Furthermore, as the flag `FCF_SHELLPOSITION` has not been set, the window will not have any data regarding its position and size on the screen.

The example of the `WM_CREATE` message allows you to learn about a way of exploiting the macros present in `PMWIN.H`. If you wish to extract parameters from the identifiers `mp1` and `mp2`, you can take advantage of an appropriate macro, and even perform a casting if necessary.

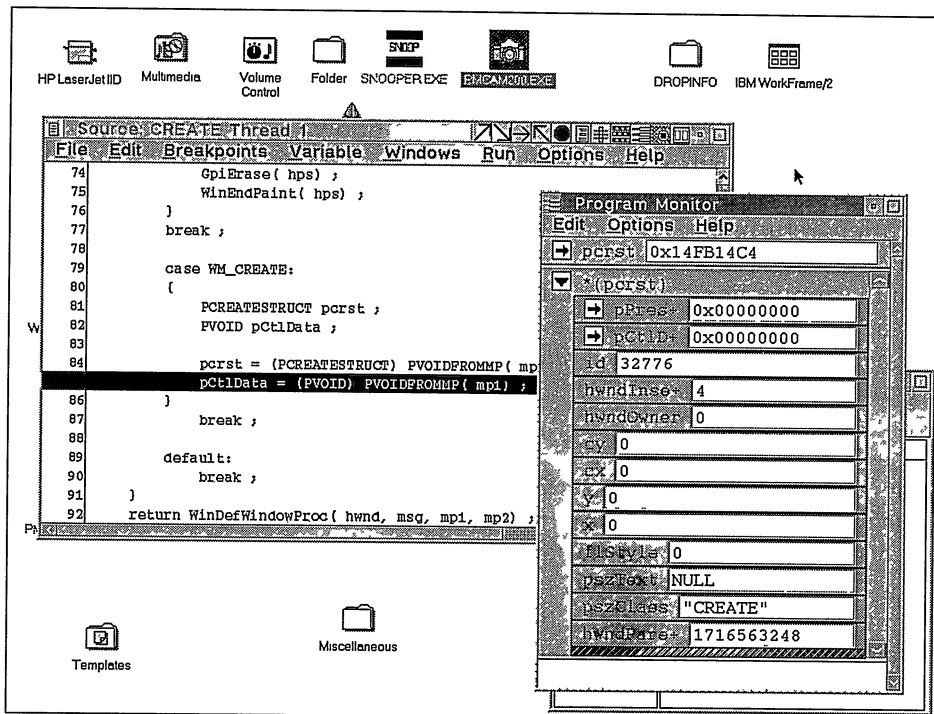


Figure 3.14 IPMD allows you to examine the memory area pointed to by the `mp2` parameter in the `WM_CREATE` message.



## Sending Messages

PM's Toolkit documentation and the on-line help explain how to retrieve the information present in `mp1` and `mp2`. Very often, however, it is necessary to send or post a message when dealing with dialog window, or in a normal window handled through its appropriate window procedure.

The passing of a message to a window procedure is often the consequence of an explicit call to the `WinSendMessage()` or `WinPostMessage()` function, or a call to one of PM's API function. Let's explore what happens in a window procedure when it receives a message. We will examine the simple case of creating a window on the screen, the client window of which will change color on two occasions:

- When the window is activated
- When the window is resized

To make the required changes to the code thus far, it is first necessary to consider your objective and determine which message is most appropriate. By looking at Appendix C, you can spot several potential candidates that could stand in a case condition and complete this exercise. The two messages that we will use are `WM_ACTIVATE` and `WM_SIZE`, respectively, for handling the window's activation and its resizing.

The `WM_ACTIVATE` message is issued by PM to a window every time the window's activation status changes via acquisition or loss. `WM_ACTIVATE` is not the only message that gets sent in this situation. It is followed by `WM_SETSELECTION` and `WM_SETFOCUS` when it is activated, and preceded by those two messages when it is deactivated.

		<i>Description</i>
<code>WM_ACTIVATE</code>	<code>0x000d</code>	
<code>mp1</code>	<code>USHORT usActive</code>	TRUE/FALSE to indicate the activation status
<code>mp2</code>	<code>HWND hwnd</code>	Handle of the frame window
Return Value	<code>ULONG flreply</code>	Reserved

For our simple application, you will intercept `WM_ACTIVATE` and see if the window is about to be activated or deactivated. The information is contained in the lower `SHORT` of `mp1` and thus can be obtained through the `SHORT1FROMMP` macro:

```

...
case WM_ACTIVATE:
    if( (BOOL) SHORT1FROMMP( mp1) )
    {
        // activation of the window
        ...
    }
    else
    {
        // deactivation of the window
        ...
    }
    break ;
...

```

Of the two possible situations in the above code fragment, we are interested only in the first one, that is, when the window is activated. Every time that execution falls through with a `WM_ACTIVATE` message and a `TRUE` value in `mp1`, we will assign a new color to the window's client area. Therefore, it is necessary to declare a `static` storage class identifier, because all output operations must be performed in the code block handling `WM_PAINT`, as optimal PM development rules dictate.

The program makes use of the `clrBck` identifier, which indicates the current color code. Once the counter gets beyond the value of `CLR_PALEGRAY (15L)`, it is restarted from `CLR_WHITE`. The switch to the next color, or the increment of the counter, is an operation performed by the code for the `WM_ACTIVATE` message. The second task is that of invalidating the entire client window of the application, thus forcing the issuing of a `WM_PAINT` message with an update region corresponding to the entire window:

```
...
case WM_ACTIVATE:
    if( (BOOL) SHORT1FROMMP( mp1))
    {
        // window is activated
        (clrBck > 14) ? clrBck = CLR_WHITE : clrBck++ ;
        WinInvalidateRect( hwnd, NULL, FALSE) ;
    }
    break ;
...

```

By calling `WinInvalidateRect()` you indicate that a `WM_PAINT` message will be delivered to the application. The logic governing a `WM_PAINT` message prescribes that the `WinFillRect()` function be called to paint the invalidated area (as a direct consequence of a resizing or activation).

```
...
case WM_PAINT:
{
    HPS hps ;
    RECTL rc ;
    hps = WinBeginPaint( hwnd, NULLHANDLE, &rc) ;
    WinFillRect( hps, &rc, clrBck) ;
    ...
    WinEndPaint( hps) ;
}
break ;
...

```

Whatever value is given as the second parameter (the address of a `RECTL` structure), `WinFillRect()` will paint only the invalidated area. This information is characteristic of a presentation space, and will be used by `WinFillRect()` without evaluating the rectangle described by the second parameter. This behavior cannot be changed by any of PM's API calls. The possible querying of the client window's size through

`WinQueryWindowRect()` and the passing of the resulting rectangle to `WinFillRect()` will always override the value of the second parameter of the fill function.

These considerations force us to follow a precise plan when implementing the code handling any `WM_PAINT` issued by the system, but is not due to any direct resizing or activation action. In fact, it might happen that some other window will overlap our application window. When the overlapping window is removed—and thus the status of the underlying window is in no way modified directly—the pixels that were previously covered become invalidated. The application will therefore receive a `WM_PAINT` message with an update region that corresponds exactly to the rectangle that was previously hidden (Figures 3.15 and 3.16).

In this case, the invalidated area is less than the size of the client window, and it is necessary to perform the repaint with the previously used color. The change of the background color is calculated outside the code handling the `WM_PAINT` message, and in the code handling the `WM_ACTIVATE` and `WM_SIZE` message with a statement like the following one:

```
...
(cClrBck > 14) ? cClrBck = CLR_WHITE : cClrBck++ ;
...

```

The `WM_SIZE` message is received when the window is resized in any direction.

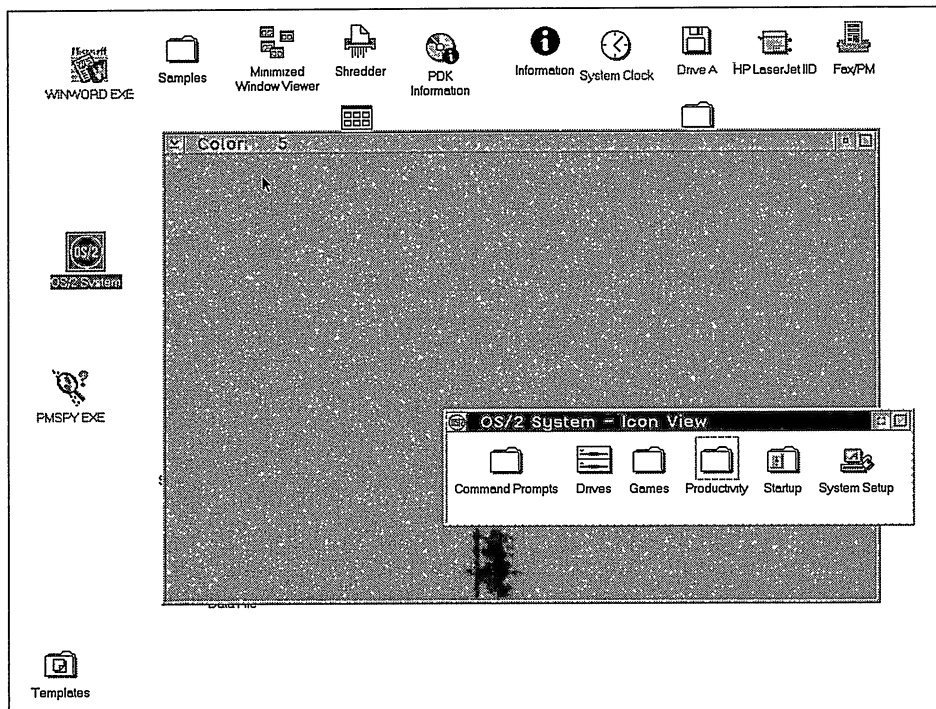


Figure 3.15 The CLIENTCL window is partially covered by OS/2 System.

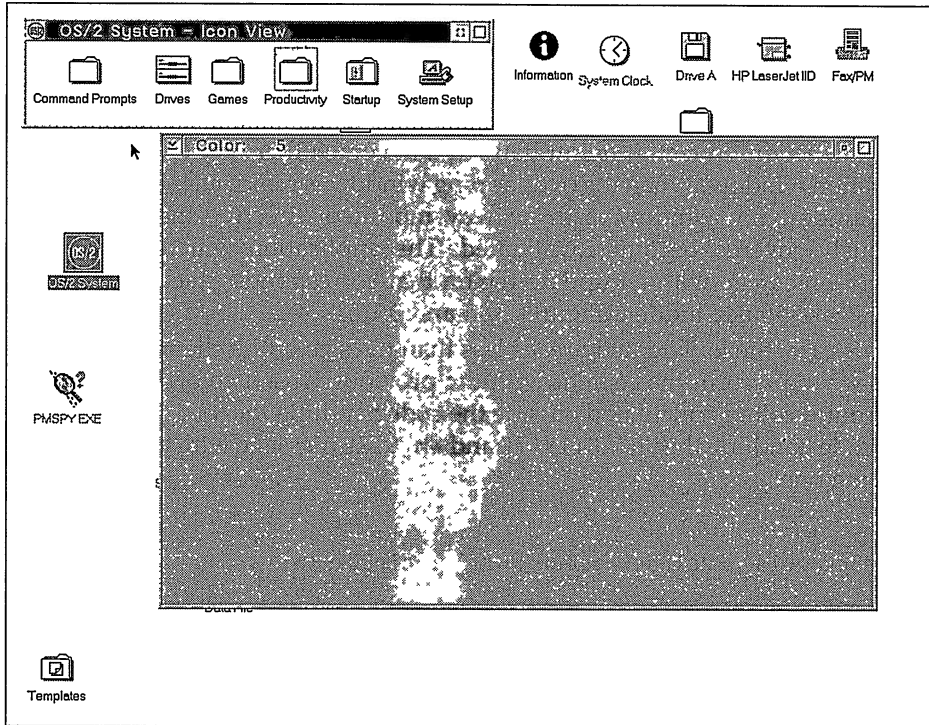


Figure 3.16 When the OS/2 System window is moved, CLIENTCL receives a WM\_PAINT message with an invalidated area that corresponds to the covered rectangle.

WM_SIZE	0x0007	<i>Description</i>
mp1	SHORT scxold	Previous size along the X axis
	SHORT scyold	Previous size along the Y axis
mp2	SHORT scxnew	Current size along the X axis
	SHORT scynew	Current size along the Y axis
Return Value	ULONG flreply	Reserved



However, as far as WM\_SIZE is concerned, it is redundant to invalidate the client window because the class to which the window belongs has the CS\_SIZEREDRAW flag set. This means that the entire client window will be invalidated automatically for any type of resizing operation it might be exposed to. Listing 3.6 shows the source code.

The functionality of the application shown in Listing 3.6 is certainly limited, however, if you compare it to the very first PM application, it is more complex. What you should notice is the very limited amount of code that was needed to change the simple program framework into a “personalized” application.

## Execution of Painting

We will conclude the analysis of messages and painting with an example that summarizes the two concepts. The PAINT application will display in the client window three text strings, as shown in Figure 3.17.

Changing the size of the window will not vary its contents. However, if you press the right mouse button anywhere inside the client window, you will notice a change. The text Milan is replaced with Paris. There is a Paris in Texas, but not in Italy! So we certainly have a problem in Figure 3.18 when it asks if you have ever been to Milan, France.

Actually, this error is intentional. Pressing the right mouse button generates the message WM\_BUTTON2DOWN, which is caught in the window procedure. In this piece of code the second and third strings are replaced:

```
...
strcpy( szString[ 1], "Paris,") ;
strcpy( szString[ 2], "France") ;
...
```

and part of the client window is invalidated. To be more precise, the rectangle containing the second string (Milan) is invalidated before it is replaced. The call to `WinInvalidateRect()` issues a WM\_PAINT into the application. The code dealing with

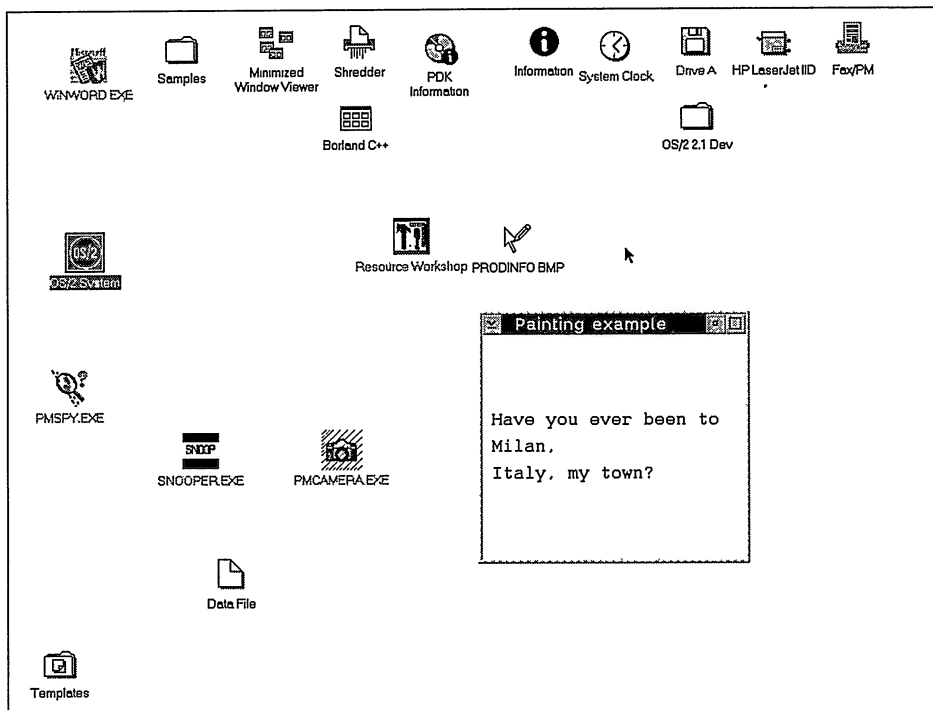
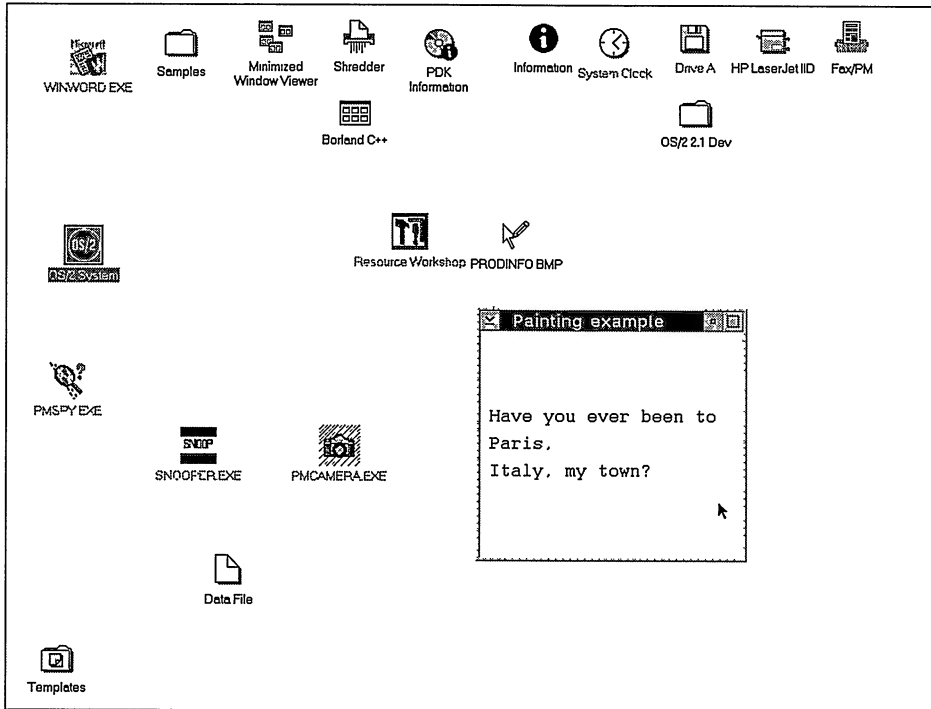


Figure 3.17 The client window of the PAINT application shows three text strings.



**Figure 3.18** Variation in the second string of PAINT's client window after the right mouse button has been pressed.

WM\_PAINT takes care of displaying the three strings call three times the *GpiCharStringAt()* function.

```

...
GpiCharStringAt( hps, &pt, sizeof( szString[ 0] ) szString[ 0] ) ;
...
GpiCharStringAt( hps, &pt, sizeof( szString[ 1] ) szString[ 1] ) ;
...
GpiCharStringAt( hps, &pt, sizeof( szString[ 2] ) szString[ 2] ) ;
...

```

You might thus expect to see the three strings as they should appear after pressing the right mouse button. However, that does not happen (Figure 3.18). The reason is to be found in the behavior of the functions that perform output on the screen: *output clipping*.

All of the *Gpi* functions, and some of the *Win* functions like *WinFillRect()* take as their first parameter a handle to a presentation space. The *hps* returned by *WinBeginPaint()* is aware of the update region. This means that the function generates its output when it falls within the invalidated area, and thus avoids any changes to the screen if the output surface is not affected by the change. This is the reason why the word



“France” appears on the string after a right mouse button click. The first call to `GpiCharStringAt()`, which handles the string “Have you ever been to,” is entirely outside the invalidated area and thus a self-censorship condition is induced, whose only purpose is that of optimizing the system’s performance. The same criterion is also applied to the third text string, even if it is changed with respect to the previous screen display. Internally, in the system’s memory, you will have France; but on the screen the string “Italy, my town?” will remain. In the case of the second string, there is a complete match between the invalidated area and the output surface, and thus the associated text is displayed. This is why you see “Paris”. The source code of PAINT (Listing 3.7) illustrates the output clipping mechanism of PM.

In order for the output of the PAINT window to correspond to what is actually in the computer’s memory, it is enough to change the size of the window. This operation invalidates the entire client window, and thus all three strings get refreshed.

This sample PAINT has been presented to clarify the behavioral model followed by the functions of PM that generate output. This choice is made to minimize CPU time dedicated to performing output operations. Figure 3.19 shows PAINT after it has been resized.

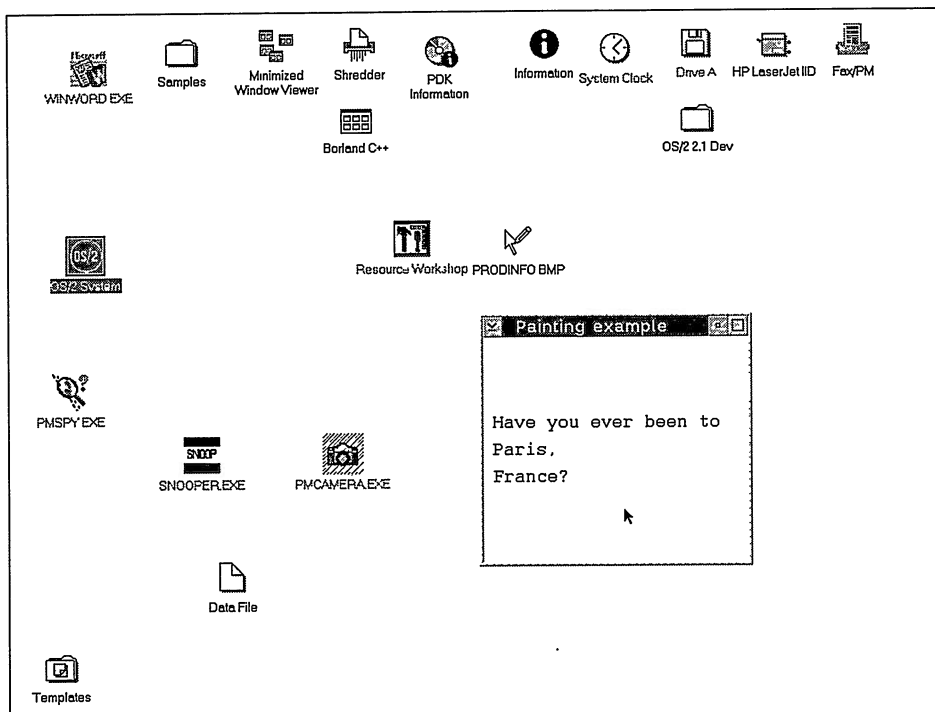


Figure 3.19 The new writing in PAINT after a complete resizing of the window.





# Windowing

We started using the *WinCreateStdWindow()* function to create a window in Chapter 2. Actually, as you might have seen in Chapter 3 by inspecting the source code with IPMD, the *WinCreateStdWindow()* not only creates a simple window, but several windows simultaneously. We know about a window called the *frame window*, identified by the handle returned by the *WinCreateStdWindow()* function and we know about the handle that identifies the *client window*, which is specified as the last parameter in the same *WinCreateStdWindow()* function. However, the final outcome of using *WinCreateStdWindow()* is much more elaborate and complex. Usually there will be a set of five or six windows, including the *frame window* and the *client window*. First we will analyze exactly what happens when the *WinCreateStdWindow()* function is called.

---

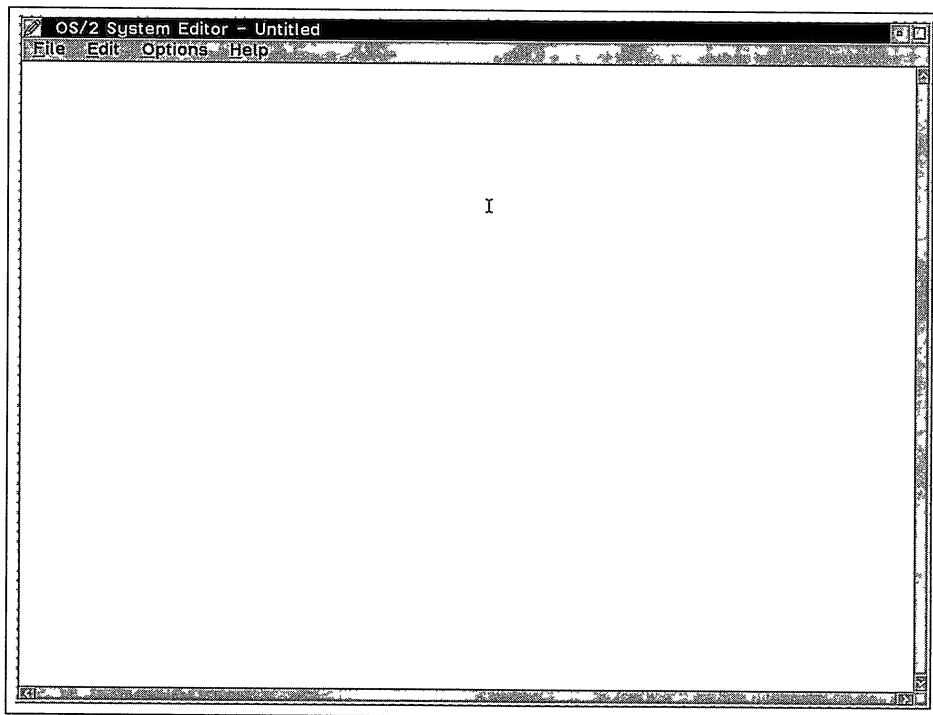
## Creating a Window with *WinCreateStdWindow()*

The effect of *WinCreateStdWindow()* as it has been used so far, is that of simultaneously creating five distinct windows, as shown in Figure 4.1. These five windows are called the *frame window*, the *titlebar window*, the *titlebar menu*, the *sizing icon*, and the *client window*. The user will see one single thing, with no direct indication of the various components. For instance, a double-click on the titlebar will maximize the window. A second double-click on the maximized window will restore its previous size and position. Let's now examine in detail the properties of a window created by *WinCreateStdWindow()*.

---

## The Frame Window

The return value of *WinCreateStdWindow()* is a handle to the frame window (*hwndFrame*). PM contains a class of predefined windows, *WC\_FRAME*, to which all standard frame windows created in application programs belong. Even what is created by the *WinCreateStdWindow()* function belongs to this class. Looking at Figure 4.1 it is quite difficult to discern the frame because it does not correspond to any distinctive and visible item of the window. (Actually, only the sizing border can be thought of as the outermost portion of the frame window.) The purpose of the frame



**Figure 4.1** A typical PM window with its various components revealed.

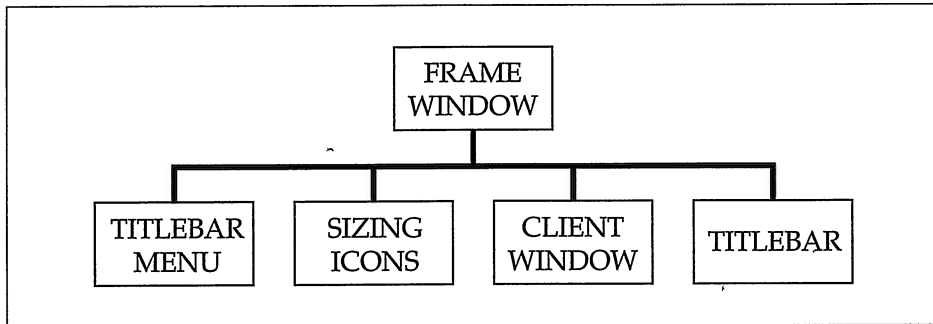
window, is that of coordinating all other windows seen as a single window on the screen by the user. Furthermore, the frame provides a screen area within which all these items are contained. Figure 4.2 shows the relationship between the frame window and the other four windows produced automatically by the *WinCreateStdWindow()* function.

It is clear that there is a hierarchical relationship among the frame window and the other windows created by *WinCreateStdWindow()*. The frame window acts as the parent and supervisor of the other four windows. This allows us to introduce two of the most fundamental concepts in PM programming, which will help clarify the relationship among the various windows created by the *WinCreateStdWindow()* function. These two concepts are those of *parenthood* and *ownership*.

---

## Parenthood

Every window produced in the PM screen group is a child window of another window. This is an absolute rule. Therefore, it is always possible to find any window's parent window, knowing that the originator is represented by the screen's background, known as the *desktop* window (defined in PMWIN.H as *HWND\_DESKTOP*). From



**Figure 4.2** Relationship scheme between all windows created by the *WinCreateStdWindow()* function.

a strictly theoretical point of view the ancestor window is `HWND_DESKTOP`; but really, the desktop of a PM screen group is visible only briefly when the system is booted. In fact, almost immediately the screen is entirely covered by the Workplace Shell, which appears as a slightly modified window belonging to the `WC_CONTAINER` class. The user cannot move this window, and thus it is impossible to access the `HWND_DESKTOP`, even by using such “intelligence” tools as `PMSPY.EXE` or `SNOOPER.EXE` (the first one is found in the OS/2 2.1 Development Toolkit and the second is presented in Chapter 7). From the point of view of programming, however, it is correct to refer to the definition of `HWND_DESKTOP` when you need to create a window to identify a PM program.

The window shown in Figure 4.1 is the system editor. The first parameter to *WinCreateStdWindow()* has been set to `HWND_DESKTOP` to indicate that the parent window should be the desktop. All the desktop’s child windows are known as *top-level windows*; therefore, the frame window of the window that appears in Figure 4.1 falls into this category.

The parent/child relationship is fundamental for all programming purposes in PM, because it is always the parent window that provides the pixels to the various child windows. The on-screen space occupied by a top-level window, like a frame window, is delivered by the parent window, the desktop. The *parenthood* relationship is a crucial condition that allows a window to exist on the screen by providing a certain amount of pixels. This pixel-providing relationship between a parent window and a child window is extended further even to lower hierarchical levels, including that one between the desktop and the top-level windows, as we will see shortly with frames.

The frame window behaves unusually, as it owns a portion of the screen, without using most of it. Instead, it delivers this area to the other four windows created with *WinCreateStdWindow()*. The exact number of windows created by this function depends on the style flags that have been set by the programmer: four is typical of a PM window.

The four windows generated by *WinCreateStdWindow()*—titlebar, titlebar menu, sizing icon, and client window—are effectively all child windows of the frame

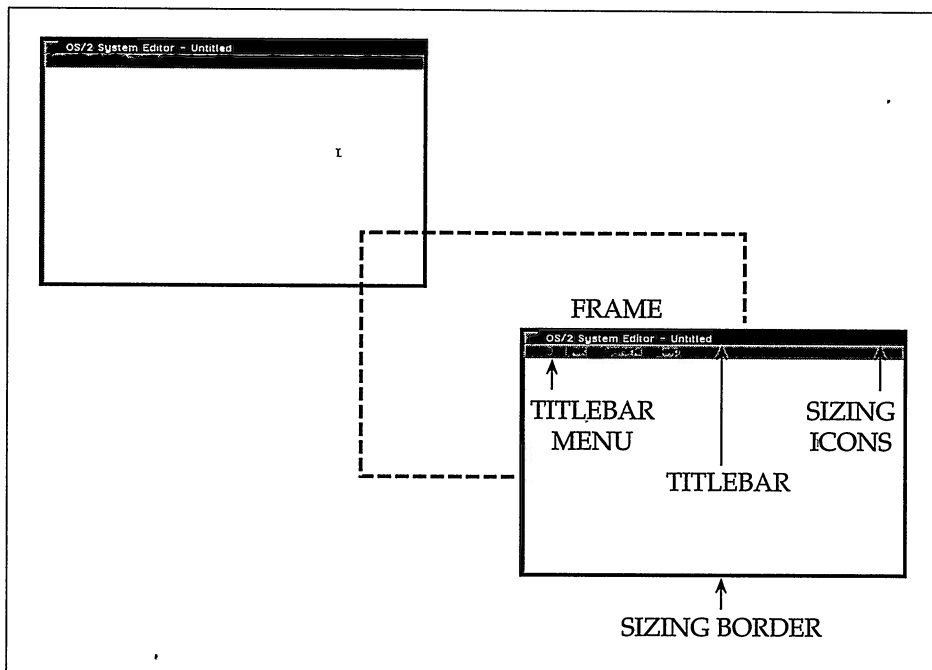
window, and thus they hold a second generation relationship with respect to the system's desktop. The relationship existing between a frame window and its child windows is depicted in Figure 4.3.

The `WinCreateStdWindow()` function allows you to establish the degree of parenthood existing between the various windows by means of the value given to its first parameter. Up to this point, we have only created top-level windows, that is, child windows of the desktop. It is nonetheless possible to specify any valid handle—referring to any other window—as the first parameter to generate a frame window and all its controls. A typical example is creating a new frame window whose parent is a client window of another window that has already been generated with `WinCreateStdWindow()`.

The use of `WinCreateStdWindow()` with the `FCF_STANDARD` flag, from which the `FCF_MENU`, `FCF_ICON`, and `FCF_ACCELTABLE` have been subtracted, will create the window's remaining structural elements, which are in turn children of the frame window.

The client window handle will thus identify a child window, and corresponds effectively to the sole object that the programmer can manage directly without having to resort to any kind of intervention on the normal flow of messages. The client window is the only window over which the developer has complete control.

With this understanding, registering a class that forms the basis of multiple-window structures becomes much clearer. When you use `WinRegisterClass()`, you actually give PM the address of a *window procedure* created by the programmer. Specifically, you



**Figure 4.3** Relationship between a frame window and its child windows.

inform PM of the window procedure that refers to the application's client window, and to that window only. All other constituent elements of a window, as they appear on the screen, really belong to other predefined windows classes in PM and access window procedures of their own, physically located in other modules of PM.

Class registration is thus a vital operation, as it is the only means of providing an application with a client window that is the only legitimate receiver of the messages generated by the user/application interaction. Listing 4.1 shows how to build two windows with a parent/child relationship.

To get to know the default IDs assigned by the frame window to its child windows, you have to look in PMWIN.H at the defines that have the FID\_ prefix, summarized in Table 4.1. and Figure 4.4.

The order of the IDs in Table 4.1 is significant for the way the application will behave. After any resizing operation, the frame window will arrange its child windows starting with the first ID. The API of PM contains a very handy function that provides the handle of any child window starting with its ID.

```
#define INCL_WINMESSAGEGR
HWND APIENTRY WinWindowFromID( HWND hwndParent, ULONG id );
```

<i>Parameter</i>	<i>Description</i>
hwndParent	Handle of a window that has child windows
id	ID of a window
<i>Return Value</i>	<i>Description</i>
HWND	Handle of the child window corresponding to that ID or NULL-HANDLE in case of error

The handle identifies the parent window of the child window you're trying to find; the ID refers to the child window. In the case of a frame window, you can obtain the handle of the corresponding window by indicating one of the IDs in Table 4.1. Here is how to get the handle of the titlebar menu of a window:

**Table 4.1 List of IDs Used by the Frame Window to Identify the Windows That Act Like Frame Controls**

<i>Default IDs for the Frame Window</i>	<i>Value</i>	<i>Description</i>
FID_SYSMENU	0x8002	Titlebar's icon
FID_TITLEBAR	0x8003	Titlebar's ID
FID_MINMAX	0x8004	Sizing icon's ID
FID_MENU	0x8005	Possible menu bar's ID
FID_VERTSCROLL	0x8006	Possible vertical scrollbar's ID
FID_HORZSCROLL	0x8007	Possible horizontal scrollbar's ID
FID_CLIENT	0x8008	Client window's ID

```

...
hwndSysMenu = WinWindowFromID( hwndFrame, FID_SYSMENU ) ;
...

```

*WinWindowFromID()* operates with any pair of parent/child windows, and is very useful even when dealing with *dialog windows* (Chapter 8). By knowing the dialog window's handle and the ID of any of its controls, you can get to that control's handle. Quite often, in addition to *WinWindowFromID()*, you can use *WinQueryWindow()* to obtain further information associated with a particular window.

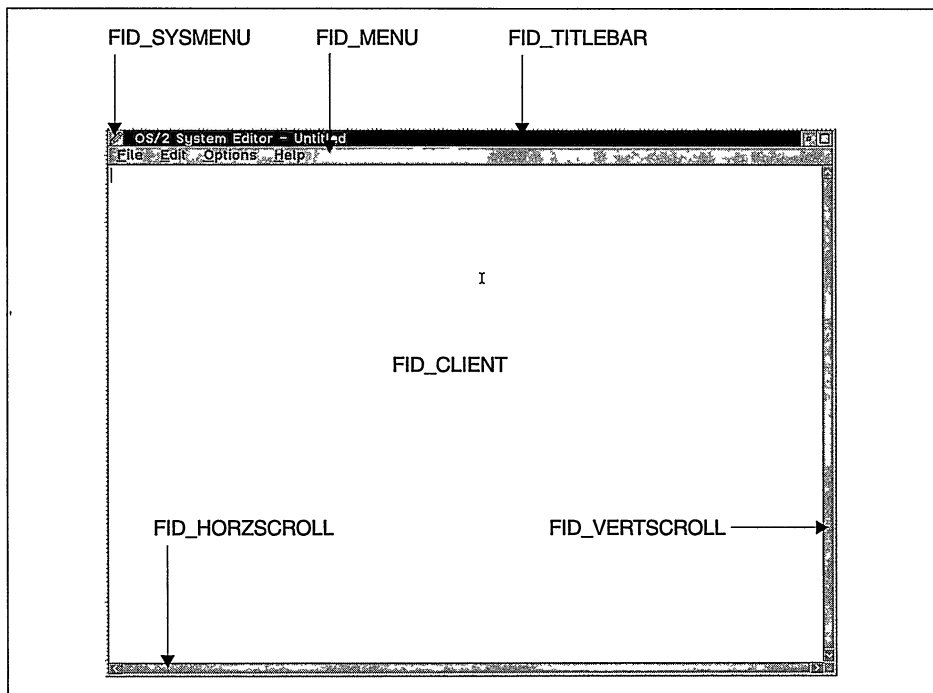
```

#define INCL_WINMESSAGEMGR
HWND APIENTRY WinQueryWindow( HWND hwnd, LONG cmd ) ;

```

<i>Parameter</i>	<i>Description</i>
hwnd	A window handle
cmd	Flag to indicate the item sought for
<i>Return Value</i>	<i>Description</i>
HWND	Handle of the window being sought for or NULLHANDLE in case of error

After indicating the handle of the window about which you want to know more, it is necessary to provide one of the indexes defined in PMWIN.H. Table 4.2 lists all valid indexes for this function.



**Figure 4.4** List of IDs assigned by the frame window to each of its child windows.

Table 4.2 Flags Accepted by *WinQueryWindow()*

<i>Flag</i>	<i>Value</i>	<i>Description</i>
QW_NEXT	0	Returns the window that comes immediately after the one indicated.
QW_PREV	1	Returns the previous window, which is also above the window indicated.
QW_TOP	2	Returns the handle of the foreground window.
QW_BOTTOM	3	Indicates the bottom-most child window, according to the screen's arrangement.
QW_OWNER	4	Returns the handle of the owner window.
QW_PARENT	5	Returns the handle of the parent window.
QW_NEXTTOP	6	Returns the next <i>top-level</i> window, according to the arrangement managed by the system; it corresponds to the window that would be activated by pressing ALT+ESC.
QW_PREVTOP	7	Returns the previous top-level window, according to the arrangement managed by the system.
QW_FRAMEOWNER	8	Returns the owner of the window indicated by the first parameter, changed so as to share the same parent as hwnd.

Using *WinWindowFromID()* and *WinQueryWindow()* together, you can trace any frame control window starting with a client window. In fact, the handle passed by the system to the window procedure's first parameter always corresponds to the client window, as we have seen. Through *WinQueryWindow()* it is possible to get back to the client window's parent window, and thus to the frame window. From this, you get to know the handle of any other frame control that might be present. Let's imagine that the following operations take place when the WM\_CREATE message is detected:

```

...
case WM_CREATE:
{
    HWND hwndFrame, hwndMenu ;

    hwndFrame = WinQueryWindow( hwnd, QW_PARENT) ;
    hwndMenu = WinWindowFromID( hwndFrame, FID_MENU) ;
    ...
}
break ;
...

```

In this example the handle of the menu window associated with the application is being retrieved, probably because the program will need to send some message to it

eventually. The same approach can be taken with any other frame control or any other child window that possesses an ID. Therefore, IDs are crucial for child windows.

As we will see in the following chapters, using *WinQueryWindow()* and *WinWindowFromID()* together is very common. Thus, it is practical and convenient to create the following two preprocessor macros:

```
#define PAPA( x) WinQueryWindow( x, QW_PARENT)
#define MENU( x) WinWindowFromID( x, FID_MENU)
```

---

## Sibling Windows

The description of the *WinWindowFromID()* and *WinQueryWindow()* functions would not be complete without touching upon another common windowing concept—sibling windows. While the strongest relationship is that between a parent and a child window, like a frame and a client window, there is another significant relationship that can be thought of as "brotherhood" among windows that are children of the same parent. The client window and the titlebar are both generated by *WinCreateStdWindow()*, and are a typical example. We usually say that the two windows are *siblings*. The `WS_CLIPSIBLINGS` flag or the class level flag `CS_CLIPSIBLINGS` refer to the management of screen area between windows at the same hierarchical level, in the case that at least one of them is moveable and might overlap the other.

All the frame controls are children of the frame window, and thus are siblings. Often, it is helpful to traverse the family tree of a window vertically, or even horizontally, to track down all kinships. The two functions, *WinWindowFromID()* and *WinQueryWindow()*, meet this need. Starting with the handle of a generic window, the programmer has at hand all the tools needed to discover any preceding generation, and eventually get to the system's desktop. Starting with any window handle, it is possible to assess all windows present in the system at any given moment. Figure 4.5 gives the whole picture.

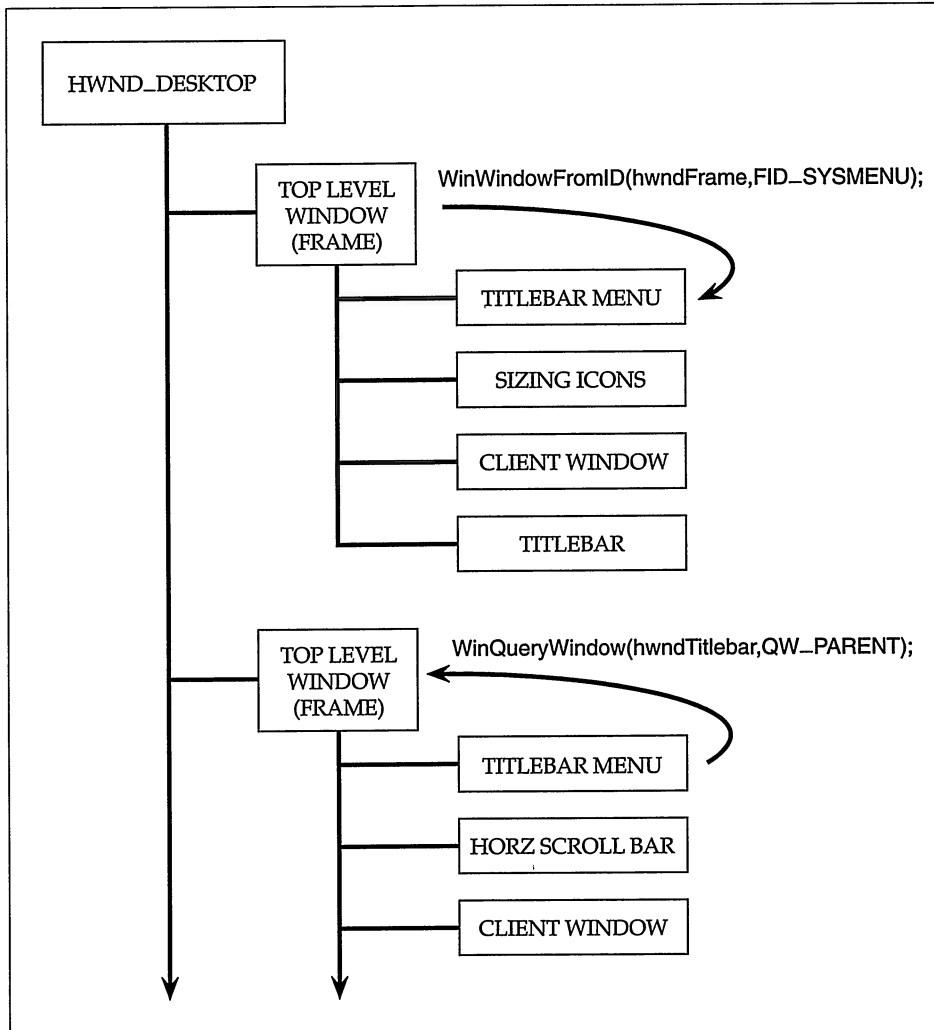
The 13th century's Italian poet Dante Alighieri stated in a famous verse of his *Divina Commedia* that "... God is the universe's Prime Mover," that is, the center of everything. Using his words in terms of PM programming it turns into "a window handle is the Prime Mover of PM programming." The keystone in PM programming is given by the handle of any window. Once you get hold of a window handle, you can control almost any aspect of PM's behavior.

---

## Ownership

The second unique feature of managing windows in PM is given by the *ownership* relationship that exists between two windows that do not necessarily have to be in a direct parent/child relationship. The ownership relationship is expressed by a notification given by a window to another window. It is the establishment of a relationship among windows that is comparable to what happens in a military structure with a





**Figure 4.5** *WinQueryWindow()* and *WinWindowFromID()* are used for examining the list of windows present in PM.

rigid reporting mechanism. In the *WinCreateStdWindow()* function there is no specific provision for defining which window should be the supervisor of the windows that will be created. This means that all of the frame's child windows will consider it their owner, with the exception of `FID_CLIENT`. The titlebar, the titlebar menu, the menu bar, and the sizing icons are all instances of predefined classes. In their respective window procedure there is some kind of notification logic implemented by the system's designers that refers to their owner. In the case of a client window's window procedure, it is up to the software designer to establish whether this behavioral model should be followed.

Each of the frame's child windows will notify any activity to their owner—the frame—thus allowing it to take any appropriate action. To clarify what the ownership relationship means, let's look at an example of window management. When you move the mouse over any of the window's sizing borders, press the left mouse button, and drag the mouse pointer, you will resize the window. Now that we know about the relationship that exists between the frame window and the four child windows, it is evident that the movement of any of the window's borders must be notified to the frame, because it must be able to adapt itself to the screen area being covered. (This means that the frame must request a greater number of pixels.) By being notified about the new size, the frame will take care of repositioning of all of its child windows, while maintaining the original structure and layout.

Another example: A double-click on the titlebar generally indicates that the user wants to maximize the window. Actually, the physical action of detecting the mouse input takes place on a window belonging to the `WC_TITLEBAR` class. This window will be responsible for notifying its owner about the action. The frame will in turn react to this by seizing the whole screen area, repositioning its controls so as to cover the whole of the available area.

The `WinCreateStdWindow()` function will not indicate an owner when a window is being created; but this problem is solved by the `WinSetOwner()` function. By default, a top-level window produced with `WinCreateStdWindow()` has no owner.

The syntax of `WinSetOwner()` is the following:

```
#define INCL_WINMESSAGEGR
BOOL WINAPI WinSetOwner( HWND hwnd, HWND hwndNewOwner) ;
```

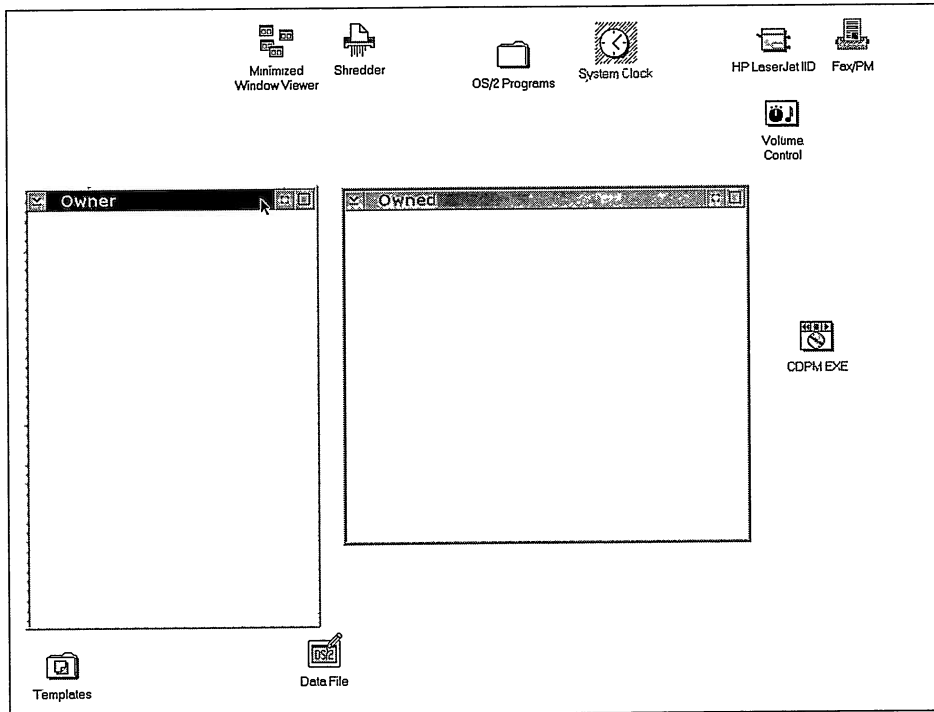
<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window whose owner you wish to refer to
hwndNewOwner	Handle of the new owner window
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The first handle indicates the window being referenced by the action of ownership change; the new owner is indicated by the `hwndNewOwner` parameter. In order to force a client window to have its frame window as its owner, you have to proceed in the following way:

```
....
WinSetOwner( hwndClient, hwndFrame) ;
...
```

You can put these concepts into practice at once by creating two windows belonging to two different classes, and naming them `Owner` and `Owned`. Immediately after the creation of the second window, you change its owner by assigning the frame window of the first window. Once the windows are displayed (Figure 4.6) you will see the effects of ownership.

The first consequence of this relationship is obvious whenever you move the `Owner` window. A similar movement will take place automatically for the `Owned` window



**Figure 4.6** Two top-level windows related by ownership.

so that it will maintain unchanged its relative position with respect to its owner. There is no mystery about this; it is simply the result of the passing of the WM\_OWNER-POSCHANGE message, issued by the owner window whenever there is a change in its position on the screen.

		<i>Description</i>
WM_OWNERPOSCHANGE	0x0052	
mp1	PSWP pswp	Address of a SWP structure that the receiver will have to change by indicating its new position and size, with respect to the corresponding variation in its owner
mp2	PSWP pswpOwner	Address of a SWP structure that contains the size and position of the owner
Return Value	ULONG flreply	Reserved

The complete message sequence received by the owned window will contain the following messages whenever there is a movement of its owner:

```
WM_OWNERPOSCHANGE
WM_ADJUSTWINDOWPOS
```

```

WM_QUERYTRACKINFO
WM_QUERYBORDERIZE
WM_QUERYBORDERIZE
WM_MOVE
WM_WINDOWPOSCHANGED

```

Furthermore, whenever the owner window is minimized, a corresponding action will occur in the owned window. Even more interestingly, in the *Minimized Window Viewer* folder you will see only the icon of the owner. Naturally, none of the aforementioned behavior will take place if the owned window is to be moved, restored, or minimized. Listing 4.2 shows the source code of this example.



The subordination relationship of the owned window with respect to the position and size of the owner window is a default property of the ownership relationship. In order to avoid this, the owned window only has to set the `FCF_NOMOVEWITHOWNER` flag; it will then be able to move freely in relation to its owner.

An OS/2 2.1 application that takes advantage of the ownership relationship among windows is the *Glossary*. Whenever a subject is selected, a new window will be displayed on the screen, at the right-hand side of the application. Any movement of the *Glossary* will be reflected automatically even on its subordinate window.

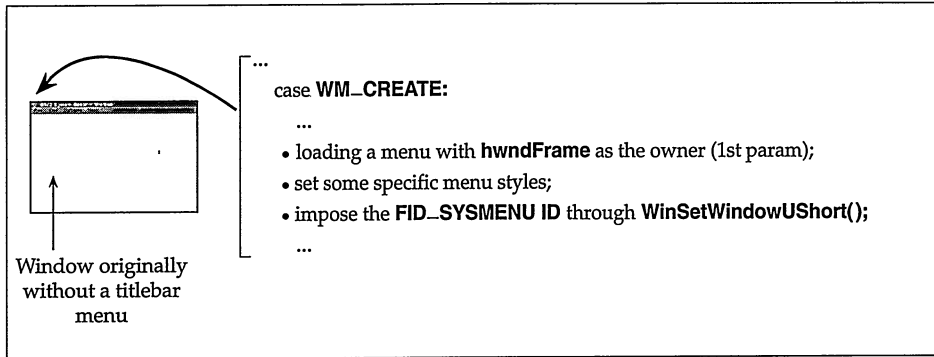
Yet another source of examples of ownership is given by the development of dialog windows, a subject that will be investigated in Chapter 8.

## The Frame Control Window

The presence of frame controls is governed by the `FCF_` flags specified as the third parameter to the `WinCreateStdWindow()` function. This API is a very simple and practical way to obtain a complete window, but it can be partitioned into its various basic components to better satisfy the software's design requirements.

In certain cases, however, you will have to create a window according to some application-specific criteria, and you might want to manage it like a frame control window, in order to gain all the advantages of this condition. We have already stressed how the titlebar menu of an OS/2 window is quite variable, as a direct consequence of the various design interpretations like CUA 89, CUA 91, or WPS. The titlebar menu produced by PM's API as a result of a call to `WinCreateStdWindow()` adheres to what is called for by CUA 89 as well as by CUA 91. However, the generation of the titlebar menu that is induced by the `FCF_SYSMENU` flag in `WinCreateStdWindow()` is not always welcome; for example, if you want to create a menu more similar to those of WPS objects.

To do this, you will have to generate the window without the `FCF_SYSMENU` flag, so that the titlebar menu will not be created. (It is also possible to destroy a frame control once it has been created and then replace it with another, new window. In any case, the operations to perform are equivalent.) You can then proceed by creating a window with the `WinCreateWindow()` function (which will be examined shortly) or by other means. After that, you need only to insert the new object into the window, and



**Figure 4.7** Scheme of creation of a window like a *frame control* and its subsequent activation.

assign it the role of the titlebar menu; to do this, you will need to act on setting the parent or owner and assign an appropriate ID (Figure 4.7).

Remember that all frame controls with the exception of `FID_CLIENT` will have as their parent and owner the frame window. When you are dealing with the titlebar menu, this situation might be set with a pair of functions: `WinSetParent()` and `WinSetOwner()`.

```

#define INCL_WINMESSAGEGR
BOOL APIENTRY WinSetParent( HWND hwnd,
                           HWND hwndNewParent,
                           BOOL fRedraw ) ;

```

<i>Parameter</i>	<i>Description</i>
<code>hwnd</code>	Handle of a window the parent of which you wish to change
<code>hwndNewParent</code>	Handle of the new parent window
<code>fRedraw</code>	Flag to indicate any possible refresh of the windows involved in the change of parent

<i>Return Value</i>	<i>Description</i>
<code>BOOL</code>	Success or failure of the operation

The third parameter, if set to `TRUE`, implies the execution of all redrawing operations, if needed, in the windows indicated by the two handles. The assignment of the owner is through `WinSetOwner()`, as previously described. PM does not provide any specific function to set an ID, but you can resort to the more generic and flexible `WinSetWindowUShort()`. The last operation is that of assigning an appropriate ID to the window that will have the role of frame control window. Table 4.1 lists all appropriate values.

```

#define INCL_WINWINDOWMGR
BOOL APIENTRY WinSetWindowUShort( HWND hwnd,
                                  LONG index,
                                  USHORT us ) ;

```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of a window the parent of which you wish to change
index	Position where to store a USHORT
us	Two-byte value to be stored
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

---

## Reserved Memory

Every window has a piece of reserved memory that stores all data that identifies and qualifies itself. When a window is created, a block of memory is allocated internally in the PM modules, just like we have seen in Chapter 2 for window classes. This reserved memory area is accessible to applications only by using appropriate PM API functions, and represents a somewhat unusual system programming domain, because not all relevant information is documented. Table 4.3 lists the functions that act upon the reserved memory area of each single PM window.

The overall scheme of the reserved memory area of each window clarifies why this storage area is useful (Figure 4.8).

The portion below the horizontal line represents the private area of each window; it is this area that can be accessed only through the functions listed in Table 4.3, by specifying appropriate indexes. These values are referred to as *magic cookies*, simply because they are quite often negative indexes, as the following portion of PMWIN.H shows (Table 4.4).

**Table 4.3 Functions for Accessing Data Present in the Reserved Memory Area of Each PM Window**

<i>Function</i>	<i>Description</i>
WinQueryWindowUShort()	Retrieves information from a pair of bytes in the reserved memory area of a window.
WinSetWindowUShort()	Sets information from a pair of bytes into the reserved memory area of a window.
WinSetWindowULong()	Sets information from four bytes into the reserved memory area of a window.
WinQueryWindowULong()	Retrieves information from four bytes in the reserved memory area of a window.
WinSetWindowPtr()	Sets a pointer to the reserved memory area of a window.
WinQueryWindowPtr()	Retrieves a pointer to the reserved memory area of a window.

---

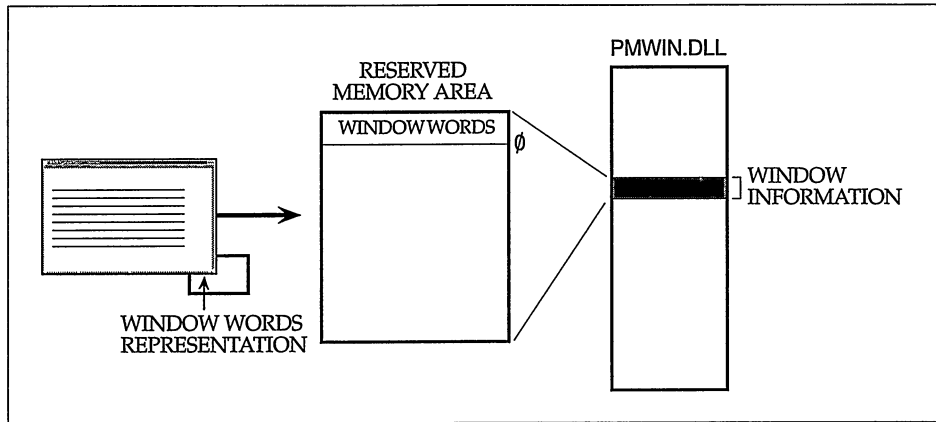


Figure 4.8 Scheme of reserve memory area of each PM window.

Table 4.4 Indexes Used to Access Information in the Reserved Memory Area of a PM Window

Flag	Value	Description
QWS_USER	0	Area available to the programmer.
QWS_ID	(-1)	Window ID.
QWS_MIN	(-1)	Position occupied after being minimized.
QWL_USER	0	Area available to the programmer.
QWL_STYLE	(-2)	Window styles.
QWP_PFNWP	(-3)	Address of the window procedure associated with this window.
QWL_HMQ	(-4)	Handle of the message queue.
QWL_RESERVED	(-5)	Reserved.
QWL_MIN	(-6)	Position occupied after being minimized.
QWL_HHEAP	0x0004	Handle to the heap (obsolete).
QWS_FLAGS	0x0008	Frame or dialog state indicator.
QWS_RESULT	0x000a	Value set by <i>WinDismissDlg()</i> .
QWS_XRESTORE	0x000c	Restore position along the X axis.
QWS_YRESTORE	0x000e	Restore position along the Y axis.
QWS_CXRESTORE	0x0010	Restore size along the X axis.
QWS_CYRESTORE	0x0012	Restore size along the Y axis.
QWS_XMINIMIZE	0x0014	Minimize position along the X axis.
QWS_YMINIMIZE	0x0016	Minimize position along the Y axis.
QWL_DEFBUTTON	0x0040	Default button in a dialog window.

It is not possible to fully describe the contents of the memory area associated with a window when that window is created, because it is a portion of the system that could or should be subject to some structural changes in forthcoming releases. Therefore, it is strongly advisable to access to this area exclusively through the magic cookies defined in PMWIN.H, and avoid using any random values that might not be documented at all.

Let's assume we need to know which style flags are associated with a particular window. This information takes up four bytes in *WinCreateStdWindow()* as well as in *WinCreateWindow()*, as we will see shortly. For this, you have to use the *WinQueryWindowULong()* function to retrieve all the style flags.

```
#define INCL_WINWINDOWMGR
ULONG APIENTRY WinQueryWindowULong( HWND hwnd, LONG index) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Window handle
index	Position of the reserved memory area to which you need to access
<i>Return Value</i>	<i>Description</i>
ULONG	4 byte information containing the data requested with the second parameter

The first parameter picks out the window to investigate, and the second parameter is the index into the reserved memory area allocated by PM when the window was created. The return value contains the four-byte information requested. If, for instance, you were interested in assessing the setting of the WS\_VISIBLE flag, and setting it if it were turned off, you would have to write:

```
...
if( !(( ul Style = WinQueryWindowULong( hwnd, QWL_STYLE))
      & WS_VISIBLE))
    WinSetWindowULong( hwnd, QWL_STYLE, ulStyle | WS_VISIBLE) ;
...
```

Some of the PM API functions define a proper working logic by examining the contents of the subarea determined by the value of QWL\_STYLE. This is the case, for instance, in *WinShowWindow()* and *WinIsWindowVisible()*. The first function selectively inserts and removes the WS\_VISIBLE style flag in a window's reserved memory area. On the other hand, *WinIsWindowVisible()* returns TRUE when the WS\_VISIBLE flag is turned on. In practice, the *if* statement of the previous example is essentially equivalent to calling *WinIsWindowVisible()*.

In Figure 4.8, an additional memory area is assumed to be present in the portion above the horizontal line. This area corresponds to the number of bytes defined by the programmer when registering the window's class.

The operation of changing one or more bits in a four-byte object of a window's reserved memory area is often performed with the *WinSetWindowBits()* function.



```
#define INCL_WINWINDOWMGR
BOOL WINAPI WinSetWindowBits(  HWND hwnd,
                               LONG index,
                               ULONG flData,
                               ULONG flMask) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Window handle
index	Index in the reserved memory area
flData	Indication of the flags to act upon
flMask	Set of bits that need to be turned on or off
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The first parameter indicates the window involved in the operation, and the second parameter is the index in the reserved memory area. The *WinSetWindowBits()* function acts exclusively upon four-byte subareas, and includes all the QWL\_ defines and the portion of window words allocated by class registration. With the third parameter, you determine the packet of bits that need to be set, and the fourth parameter defines the kind of setting you want to impose onto the bits indicated by the third parameter. For instance, if you want to set the WS\_VISIBLE flag to TRUE, and leave unchanged any existing bit configuration, you can proceed in the following way:

```
...
WinSetWindowBits( hwnd, QWL_STYLE, WS_VISIBLE, WS_VISIBLE) ;
...
```

To sum up, the action performed by *WinSetWindowBits()* is analogous to the effect produced by *WinSetWindowULong()* and why this last function appears so often in listings.

---

## Extending the Reserved Memory Area

In addition to the ordinary data stored automatically by PM when a window is created and during its use, the programmer can also define a supplemental area that augments the standard reserved memory. This operation must take place when the window class is being registered.

```
BOOL WINAPI WinRegisterClass(  HAB hab,
                              PSZ pszClassName,
                              PFNWP pfnWndProc,
                              ULONG flStyle,
                              ULONG cbWindowData) ;
```

The fifth parameter of *WinRegisterClass()* indicates the size in bytes of the extra space of reserved memory that will be allocated to each window belonging to the class being registered. The programmer can access this additional area through the pair of functions *WinSetWindowUShort()* and *WinSetWindowULong()*, and through

the *WinSetWindowPtr()* function. Typically, such an additional area stores data closely related to the use and behavior of the window. To retrieve any of this data, you revert to *WinQueryWindowUShort()* and *WinQueryWindowULong()* and *WinQueryWindowPtr()*. The defines of *QWL\_USER* and *QWS\_USER* set in *PMWIN.H* correspond to level 0 in Figure 4.8; that is, the first available position for storing window specific information.

---

## Querying the Reserved Memory Area

The reserved memory area of each window contains several interesting pieces of data which are very useful in an OS/2 application, both from the point of view of creating a program compliant with the MDI model as well as that of SDI, which is somewhat more faithful to the CUA 91 specification (Chapter 13).

The use of the *WinQueryWindowUShort()* function also allows you to trace a window's ID—an invaluable piece of information whenever a parenthood relationship among windows is involved. On the other hand, through the *WinQueryWindowULong()* function, you can get to know the set of style flags that characterize a single window. To specify the values in the reserved area, you can use the *WinSetWindowUShort()* and *WinSetWindowULong()* functions. These two functions must be used with care, because it is easy to overwrite data critical to the window's integrity. Functions like *WinSetWindowPos()* and *WinQueryWindowPos()* perform their operations by reading a window's reserved memory area directly.

The *WinSetWindowPtr()* function is, in practice, a macro function that calls *WinSetWindowULong()* to store directly the value of a pointer to a window's reserved memory area. The syntax of this function is the following:

```
#define INCL_WINWINDOWMGR
BOOL APIENTRY WinSetWindowPtr(   HWND hwnd,
                                LONG index,
                                PVOID p) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Window handle
index	Index into the window's reserved memory area
p	Pointer to store
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The third parameter, of the type *PVOID*, contains a generic pointer. The presence of this function among the many API functions available for PM is justified solely because it makes the application's code simpler and easier to read. Furthermore, *WinSetWindowPtr()* can be useful in implementing an operation known as *subclassing* a window (discussed in Chapter 10); obviously, subclassing can also be performed by means of *WinSetWindowULong()* considering that you are always dealing with a four-byte piece of information.

---

## Window Words' Usage Rules

In general, it is a good programming practice assigning *window words* to the windows of a class only if it is strictly necessary for the inner working of an application. In theory, there are some limitations on the size of window words, although experience and common sense suggest that only two values are viable solutions: 0 and 4. Higher values are allowed, although they certainly can find in direct memory management a better implementation rather than allocating window words. At present, in fact, the windowing engine of OS/2 2.1 is still a 16-bit module, and retains some of the structural limitations connected to the old segmented memory model of Intel processors.

As we will see in the discussion of the development of MDI application, the logic behind this scheme accommodates in the window words only a pointer to any additional and separate memory area (managed by the application thanks to several of PM's API support functions for memory management). It is in this additional memory area that all relevant window data should be stored.

---

## Message Passing

The indexes listed in Table 4.4 indicate that it is possible to find the address of a window's window procedure by querying the window's handle. This is extremely important and useful in *subclassing* a window (Chapter 10). For the moment, however, let's get back to analyzing the process of registering a window class, and the criteria followed by PM in issuing messages. As you might recall, the data of a class is stored internally in the various modules that make up PM, and not in the data segment of the application that performs the registration. When we examined the mechanism adopted by *WinDispatchMsg()* for reaching the window procedure of a class, we assumed that the whole process of tracking back to the procedure's address was based on the association between a window handle and the class to which that window belonged. Only now can we really understand how PM behaves. When *WinDispatchMsg()* must jump to a window procedure, it will retrieve the window procedure's address from the window's reserved memory area—not from some place at the class level. The whole operation is fast, but this is not the most important aspect. We now have all tools available for making an informed guess as to how the code of the *WinDispatchMsg()* and *WinSendMsg()* functions really work. Let's start with the first one.

```
MRESULT APIENTRY WinDispatchMsg(   HAB hab,
                                   PQMSG pqmsg)
{
    PFNWP pfnwp ;

    // retrieving the window proc address
    pfnwp = (PFNWP)WinQueryWindowPtr( pqmsg -> hwnd, QWP_PFNWP) ;
```



```

        ULONG id,
        PVOID pCtlData,
        PVOID pPresParams) ;

```

<i>Parameter</i>	<i>Description</i>
hwndParent	Parent window handle
pszClass	Name of the class to which the window being constructed belongs
pszName	Text of the name that identifies the window
flStyle	Set of the window's style flags
x	Window's lower left-hand corner, on the X axis
y	Window's lower left-hand corner, on the Y axis
cx	Window's size on the X axis
cy	Window's size on the Y axis
hwndOwner	Owner window's handle
hwndInsertBehind	Handle of the window behind which the window being constructed should be placed
id	Window's ID
pCtlData	Pointer to the data structure pertaining to the window
pPresParam	Pointer to the windows presentation parameters
<i>Return Value</i>	<i>Description</i>
HWND	Handle of the window being constructed or NULLHANDLE in case of failure

The first parameter to *WinCreateWindow()* is exactly the same as the one required by *WinCreateStdWindow()*, that is, a handle to the parent window. The second parameter identifies the class to which the window belongs. The software designer can indicate a class name explicitly registered by the application, the name of a predefined class, or the name of a global class.

The third parameter deployed is a pointer to a text string that will appear in the window. If you believe that this corresponds to the window's title, you are mistaken! For example, in the case of an instance of a window belonging to WC\_BUTTON class, this text corresponds to the contents of a button, or to the string that appears to the right of a radio button or a check box. For WC\_LISTBOX class instance, this text does not convey any meaning, and is always replaced by a NULL pointer. The same holds true for any instances of window classes registered by the application. The fourth parameter, a ULONG identifier, contains the window styles distinguished by the WS\_ prefix and possibly by the style of the predefined classes.

With *WinCreateStdWindow()* you simultaneously create several windows. However, the main purpose of this function is creating the window's client area. On the other hand, *WinCreateWindow()* is a tool for creating one window at a time; therefore the FS\_ styles are applicable only when creating a window belonging to the WC\_FRAME class, whilst the FCF\_ flags cannot appear in any parameter of *WinCreateWindow()*.

The next four parameters allow you to set the position and size of the window on the screen. Now that you know what parenthood is all about, it is worthwhile to elaborate on our earlier discussion. All four values are expressed in the coordinates of the parent window. For top-level windows, this corresponds to the dimensions of the screen. This no longer holds for any of their child windows. For example, the position and size of the client window generated by *WinCreateStdWindow()* refers to its own frame and not to the entire screen. As a consequence, the call of *WinQueryWindowPos()* on a client window needs a subsequent call to *WinMapWindowPoints()* to find the correct on-screen location.

The flexibility of *WinCreateWindow()* is evident even when defining an owner, which might coincide with the parent (ninth parameter). Furthermore, *WinCreateWindow()* allows you to determine the position of a window with respect to all of its siblings (tenth parameter) by indicating the handles of the windows that need to be placed before it, or the defines *HWND\_TOP* and *HWND\_BOTTOM* that occupy the foremost or the bottom most position, respectively. *WinCreateWindow()* also allows to assign an ID to the window being created. The availability of an ID is handy when managing related windows, as we have seen.

The twelfth parameter, *pCtldata*, is a pointer to a data area specific to the window being constructed. As you will see in Chapter 7, several of the predefined window classes have some data structure that can be exploited during window creation. This process is advisable for any kind of window. However, it cannot be applied when you are using *WinCreateStdWindow()*. The greater flexibility and compactness of this last function is paid for in terms of a loss of control by the programmer.

## ***Building Standard Windows with WinCreateWindow()***

To fully understand how to use *WinCreateWindow()*, let's start with a simple task: obtain via this function what we have already achieved in the preceding listings with *WinCreateStdWindow()*. By studying the syntax of *WinCreateWindow()*, you will be able to see the logic that governs *WinCreateStdWindow()*.

Instead of simply calling *WinCreateStdWindow()* you will now use a much more elaborate approach, since you will have to take care of creating the frame window, and all of the child windows that characterize it. As shown in Table 2.3, every frame belongs to the *WC\_FRAME* class, which is one of the predefined classes of PM. The creation of a frame window with *WinCreateWindow()* thus turns out to be the following:

```

...
hwndFrame = WinCreateWindow(   HWND_DESKTOP, WC_FRAME,
                               NULL,
                               WS_VISIBLE | WS_CLIPCHILDREN |
                               WS_SYNCPAINT,
                               10, 10, 100, 100,
                               HWND_DESKTOP, HWND_TOP,
                               1,
                               NULL, NULL) ;
...

```

A frame window generated by *WinCreateStdWindow()* will always have one of the following *WS\_* styles: *WS\_VISIBLE* | *WS\_CLIPSIBLINGS* | *WS\_SYNCPAINT*. In addition to these *WS\_* styles, you can also indicate those that take the *FS\_* prefix (which are not present in this example). The return value will be the handle of the frame window. At this point, the same operation should be repeated for all frame controls that should appear in the window. For each of these you have to call the same *WinCreateWindow()* function. For instance, to install a titlebar, you would need a piece of code like this:

```

...
hwndFrame = WinCreateWindow(    hwndFrame, WC_TITLEBAR,
                                NULL,
                                WS_VISIBLE | WS_PARENTCLIP |
                                WS_SYNCPAINT,
                                0, 0, 0, 0,
                                hwndFrame, HWND_TOP,
                                FID_TITLEBAR,
                                NULL, NULL) ;
...

```

Now, the first problem that arises from this approach is that of defining the position and the size of each control proposed for the titlebar above. In the preceding piece of code, both the position and the size are set to zero. Any *WC\_FRAME* class window should not have the *WS\_VISIBLE* flag set, in order to avoid displaying the actual construction process. The actual display would then be taken care of by calling *WinSetWindowPos()* once all windows have become available but not yet visible.

However, it is possible to simplify the whole process and optimize the call to *WinCreateWindow()*, as far as the frame window is concerned. The *WC\_FRAME* class has at its disposal the *FRAMECDATA* data structure defined in *PMWIN.H* that helps the programmer's work and is designed specifically for passing information to a *WinCreateWindow()* through the *pCtldata* pointer:

```

typedef struct _FRAMECDATA
{
    // fcddata
    USHORT cb ;
    ULONG f1CreateFlags ;
    USHORT hmodResources ;
    USHORT idResources ;
} FRAMECDATA ;

typedef FRAMECDATA *PFRAMECDATA ;

```

The *FRAMECDATA* structure contains four members to pass information to *WinCreateWindow()* regarding the *frame control flags* and the resources that might be associated with the window. The first member, *cb*, contains the size of the data area passed to *WinCreateWindow()*, while the following *ULONG* lists one or more *FCF\_* flags. Consequently, although it is not possible to include *FCF\_* flags directly in any of *WinCreateWindow()* parameters, thanks to the *FRAMECDATA* the problem is solved in another way for a *WC\_FRAME* window.

The third and fourth members of the FRAMECDATA structure correspond to the seventh and eighth parameters of *WinCreateStdWindow()*. They indicate, respectively, the module from which the resources (menu, icon, and accelerator table) associated with the window can be retrieved, and the resources ID. Thus, the creation of a frame window also becomes the moment for defining the window's structural elements, except the client window, as shown in the following code fragment.

```

...
#define FCF_WPS      FCF_HIDEMAX | FCF_STANDARD & ~FCF_MINMAX
...
FRAMECDATA fcdata ;
...

fcdata.cb = sizeof fcdata ;
fcdata.flCreateFlags = FCF_WPS & ~FCF_MENU & ~FCF_ICON &
                      ~FCF_ACCELTABLE ;
fcdata.hmodResources = NULLHANDLE ;
fcdata.idResources = 0 ;
...
hwndFrame = WinCreateWindow(   HWND_DESKTOP, WC_FRAME,
                               NULL,
                               WS_VISIBLE | WS_CLIPCHILDREN |
                               WS_SYNCPAINT,
                               0, 0, 0, 0,
                               HWND_DESKTOP, HWND_TOP,
                               1,
                               &fcdata, NULL) ;
...

```

Once the frame window and its desired frame controls have been created, the only remaining thing that needs to be done is to provide the window with a client window so that the programmer can control the window's appearance and functionality. To do this, it is necessary that a window class be registered. *WinCreateWindow()* once again clarifies unequivocally the usefulness and the purpose of registering a window class that is limited to and acts only on the client area of a window. In the following piece of code, you see how *WinCreateWindow()* creates the client window of a window generated with that very same function.

```

...
hwndClient = WinCreateWindow(   hwndFrame, szClientClass,
                               NULL,
                               WS_VISIBLE,
                               0, 0, 0, 0,
                               hwndFrame, HWND_TOP,
                               FID_CLIENT,
                               NULL, NULL) ;
...

```





```

    {
        case WM_CREATE:
        {
            PCREATESTRUCT pcrst = (PCREATESTRUCT)mp2 ;
            PULONG pulFCF ;

            // did the creator add some class data?
            if( pcrst -> pCtlData)
            {
                // do we have to create a titlebar?
                if( pcrst -> pCtlData -> flCreateFlags & FCF_TITLEBAR)
                {
                    hwndTitlebar = WinCreateWindow( hwnd,
                                                    WC_TITLEBAR,
                                                    ... ) ;

                    ...
                }
                ...
            }
            ...
        }
        break ;
    }
    ...
}
return WinDefWindowProc( hwnd, msg, mp1, mp2 ) ;
}

```

The direct passing of data to a window procedure of a class during the creation of a window is common practice because it allows the transfer of packets of information easily, without having to resort to any kind of source file scope data structure.

## Experiments Using WinCreateWindow()



Listing 4.3 contains the source code for creating a PM window by means of two successive calls to *WinCreateWindow()*; the first call will generate the frame window, and the second call the client window. The same operation is repeated in order to produce a second window, which becomes a child window of the first one.

---

## The CREATESTRUCT Structure

When a generic window procedure receives the WM\_CREATE message, it means that a window belonging to the corresponding class has just been created. WM\_CREATE is issued as a side effect of the execution of *WinCreateWindow()* or *WinCreateStdWindow()*. This message contains in mp1 the address to the class's specific data structure and in mp2 the address to a structure of the type CREATESTRUCT. In CREATESTRUCT the system stores all information specified when a window is being created:

```

typedef struct _CREATESTRUCT
{ // crst
    PVOID pPresParams;
    PVOID pCtldata;
    ULONG id;
    HWND hwndInsertBehind;
    HWND hwndOwner;
    LONG cy;
    LONG cx;
    LONG y;
    LONG x;
    ULONG flStyle;
    PSZ pszText;
    PSZ pszClass;
    HWND hwndParent;
} CREATESTRUCT;
typedef CREATESTRUCT *PCREATESTRUCT;

```

The 13 members of this structure correspond exactly to the number of parameters in *WinCreateWindow()*: When the message is issued by *WinCreateStdWindow()*, some of these members will take the value of NULL. The information contained in *mp1* is somewhat redundant, because an equivalent pointer also exists in the CREATESTRUCT structure, *pCtldata*.

---

## How to Destroy a Window

A PM application is characterized by the simultaneous presence of several windows; a typical case is an MDI application. The user will be presented with a choice of closing one or more windows active in the application without terminating the whole program. The closure of a window, which can be set by selecting the *Close* option from the titlebar menu or by some other means implemented by the application, implies that the frame window of the corresponding window be destroyed. PM's API has for this purpose the *WinDestroyWindow()* function:

```

#define INCL_WINWINDOWMGR
BOOL WINAPI WinDestroyWindow( HWND hwnd );

```

<i>Parameter</i>	<i>Description</i>
<i>hwnd</i>	Handle to be destroyed: any kind of window
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The sole parameter of this function is the handle of a generic window, which can thus be a client window or any other window that makes up what the user sees and perceives as a window on the screen. Thanks to the parenthood and ownership relationship, it is necessary to specify the frame window's handle to cause the complete destruction of an entire screen window. Without knowing the frame

window, it is not possible to track all pixels occupied by the child windows to which a frame window would notify the order of destroying themselves.

So far, *WinDestroyWindow()* appears only in the *main()* function, immediately after the message loop, or, when the application is essentially terminated. The destruction of all resources employed is good programming practice and style. Often, however, destruction operations are performed in a window procedure, as in MDI applications. There the available handle refers to the client window rather than to the frame window; the appropriate information is easily available through *WinQueryWindow()* by setting the pertinent flag.

---

## Closing a Window in a Multi-Window Application

When the user double-clicks on the titlebar menu's icon or selects the Close option, the window's destruction sequence is triggered. The message flow that follows is rather complicated. First, a user's interaction with a titlebar menu always generates the WM\_SYSCOMMAND message, which is passed to the WC\_FRAME class's window procedure, and is thus invisible to the application. The client window procedure will receive a WM\_CLOSE message, followed by a WM\_DESTROY message. In general, neither of these two messages is detected in special case statements, and thus the task of *WinDefWindowProc()* is that of taking care of the messages' correct processing. The net result is the destruction of the window with which the user has interacted. As a consequence, however, *WinDefWindowProc()* also posts a WM\_QUIT message to the message queue. This last message will break the message loop and thus cause the program's termination.

Sometimes, this is not the desired outcome. There is a simple solution to this problem. Inside the window procedures of all classes that produce any visible window, it is necessary to trap the WM\_CLOSE message, and prevent it from reaching *WinDefWindowProc()*. In this case, the application can decide to take the default window procedure's place in processing the message. The code you have to write must call *WinDestroyWindow()*, passing the frame window handle and not the one of the client window. The distinction is subtle, but the implications are substantial. The code is as follows:

```
...
case WM_CLOSE:
    WinDestroyWindow( PAPA( hwnd) );
    return 0L ;
...
```

In this manner you can separate a window destruction from the application termination; and that's precisely what is needed in a multiwindowing application.

## Sizing and Positioning a Window

The *WinCreateStdWindow()* function does not allow the sizing of a window on the basis of any of its nine parameters. The presence of the *FCF\_SHELLPOSITION* flag delegates to the system the business of finding a place for the window on the screen and defining dimensions for it that are large enough to provide for all of its contents, but not so large as to cover the whole screen area.

On the other hand, with *WinCreateWindow()*, you define a window's position on the screen by indicating the coordinates of its lower left-hand side corner, in addition to specifying its size. Both pairs of values are expressed in screen coordinates when related to a top-level frame. For this reason, the video adapter's resolution directly affects a window's positioning on the screen.

To overcome this problem, PM offers the *WinQuerySysValue()* function that allows you to compute the number of pixels available on the X and Y axis by setting the *SV\_CXSCREEN* and *SV\_CYSCREEN* flags. Once you have these two values, you can position the window at any precise location, depending on the video adapter being used. The syntax of *WinQuerySysValue()* is:

```
#define INCL_WINSYS
LONG APIENTRY WinQuerySysValue( HWND hwndDesktop, LONG iSysValue );
```

<i>Parameter</i>	<i>Description</i>
hwndDesktop	Handle to the desktop window
iSysValue	System value being asked for
<i>Return Value</i>	<i>Description</i>
LONG	The required information

The first parameter always has to refer to the desktop window, and therefore it is replaced by the *HWND\_DESKTOP* define. In the second parameter, you specify any of the indexes listed in Table 4.5.

**Table 4.5 Values and Meanings of the Flags Accepted by *WinQuerySysValue()***

<i>Define</i>	<i>Value</i>	<i>Description</i>
<i>SV_SWAPBUTTON</i>	0	Swaps the default mouse buttons.
<i>SV_DBLCLKTIME</i>	1	Defines the time interval between successive clicks on a mouse button.
<i>SV_CXDBLCLK</i>	2	Defines the mouse's variation in position along the X axis that is acceptable when registering a double click.
<i>SV_CYDBLCLK</i>	3	Defines the mouse's variation in position along the Y axis that is acceptable when registering a double click.

(continued)

**Table 4.5 (Continued)**

<i>Define</i>	<i>Value</i>	<i>Description</i>
SV_CXSIZEBORDER	4	Returns the size along the X axis of the sizing border.
SV_CYSIZEBORDER	5	Returns the size along the Y axis of the sizing border.
SV_ALARM	6	Returns TRUE if the alarm sound is enabled.
SV_RESERVEDFIRST1	7	First reserved value.
SV_RESERVEDLAST1	8	Last reserved value.
SV_CURSORRATE	9	Defines the cursor's blink rate whenever it is displayed as an I bar.
SV_FIRSTSCROLLRATE	10	Autoscrolling delay (in ms) when using a scrollbar.
SV_SCROLLRATE	11	Scroll rate.
SV_NUMBEREDLIST	12	Undocumented.
SV_WARNINGFREQ	13	Defines the tone frequency of every beep issued together with a warning message.
SV_NOTEFREQ	14	Defines the tone of every beep issued together with a warning message.
SV_ERRORFREQ	15	Defines the tone of every beep issued together with an error message.
SV_WARNINGDURATION	16	Defines the duration of every beep issued together with a warning sound.
SV_NOTEDURATION	17	Defines the duration of every beep issued together with a note sound.
SV_ERRORDURATION	18	Defines the duration of every beep issued together with an error message.
SV_RESERVEDFIRST	19	Reserved.
SV_RESERVEDLAST	19	Reserved.
SV_CXSCREEN	20	Return's the screen's pixel count along the X axis.
SV_CYSCREEN	21	Return's the screen's pixel count along the Y axis.
SV_CXVSCROLL	22	Width of a vertical scrollbar.
SV_CYHSCROLL	23	Height of a horizontal scrollbar.
SV_CYVSCROLLARROW	24	Returns the size of a vertical scrollbar's arrow along the Y axis.
SV_CXHSCROLLARROW	25	Returns the size of a horizontal scrollbar's arrow along the X axis.
SV_CXBORDER	26	Returns the size of a window's border along the X axis.
SV_CYBORDER	27	Returns the size of a window's border along the Y axis.

*(continued)*

**Table 4.5 (Continued)**

<i>Define</i>	<i>Value</i>	<i>Description</i>
SV_CXDLGFRAME	28	Returns the size of a dialog window's border along the X axis.
SV_CYDLGFRAME	29	Returns the size of a dialog window's border along the Y axis.
SV_CYTITLEBAR	30	Returns the size of a titlebar along the X axis.
SV_CYVSLIDER	31	Returns the size of a slider along the X axis.
SV_CXHSLIDER	32	Returns the size of a slider along the Y axis.
SV_CXMINMAXBUTTON	33	Returns the size of a sizing icon along the X axis.
SV_CYMINMAXBUTTON	34	Returns the size of a sizing icon along the Y axis.
SV_CYMENU	35	Returns the height of the menu bar.
SV_CXFULLSCREEN	36	Screen dimension on the X axis.
SV_CYFULLSCREEN	37	Screen dimension on the Y axis.
SV_CXICON	38	Returns the size of an icon along the X axis.
SV_CYICON	39	Returns the size of an icon along the Y axis.
SV_CXPOINTER	40	Returns the size of the mouse pointer along the X axis.
SV_CYPOINTER	41	Returns the size of the mouse pointer along the Y axis.
SV_DEBUG	42	Debug system (TRUE) or not debug system (FALSE).
SV_CMOUSEBUTTONS	43	Number of mouse buttons.
SV_CPOINTERBUTTONS	43	Number of pointer buttons.
SV_POINTERLEVEL	44	Pointer hide level: zero visible, greater hidden.
SV_CURSORLEVEL	45	Cursor hide level: zero visible, greater hidden.
SV_TRACKRECTLEVEL	46	Makes a tracking rectangle visible (0) or hidden (>0).
SV_CTIMERS	47	Number of system timers.
SV_MOUSEPRESENT	48	Flags whether the mouse is present.
SV_CXBYTEALIGN	49	Number of pixels for horizontal alignment.
SV_CXALIGN	49	Alignment on the x axis.
SV_CYBYTEALIGN	50	Number of pixels for vertical alignment.
SV_CYALIGN	50	Alignment on the y axis.
SV_NOTRESERVED	56	Unreserved value.
SV_EXTRAKEYBEEP	57	Delay in milliseconds before keyboard beeps.
SV_SETLIGHTS	58	Sets keyboard lights when special keys are pressed.
SV_INSERTMODE	59	Insert mode on/off.

*(continued)*

**Table 4.5 (Continued)**

<i>Define</i>	<i>Value</i>	<i>Description</i>
SV_MENUROLLDOWNDELAY	64	Delay in milliseconds before showing a pull-down menu.
SV_MENUROLLUPDELAY	65	Delay in milliseconds before hiding a pull-down menu.
SV_ALTMNEMONIC	66	Undocumented.
SV_TASKLISTMOUSEACCESS	67	Mouse buttons to access Window List.
SV_CXICONTEXTWIDTH	68	Width icon text.
SV_CICONTEXTLINES	69	Number of lines of text for icons.
SV_CHORDTIME	70	Length of chord in milliseconds.
SV_CXCHORD	71	Undocumented.
SV_CYCHORD	72	Undocumented.
SV_CXMOTION	73	Undocumented.
SV_CYMOTION	74	Undocumented.
SV_BEGINDRAG	75	Mouse begin drop message (low word) and keyboard control code (high word).
SV_ENDDRAG	76	Mouse end drop message (low word) and keyboard control code (high word).
SV_SINGLESELECT	77	Mouse single selection message (low word) and keyboard control code (high word).
SV_OPEN	78	Mouse open message (low word) and keyboard control code (high word).
SV_CONTEXTMENU	79	Mouse message (low word) and keyboard control code (high word).
SV_CONTEXTHELP	80	Mouse message (low word) and keyboard control code (high word).
SV_TEXTEDIT	81	Mouse message (low word) and keyboard control code (high word).
SV_BEGINSELECT	82	Mouse message (low word) and keyboard control code (high word).
SV_ENDSELECT	83	Mouse message (low word) and keyboard control code (high word).
SV_BEGINDRAGKB	84	Keyboard message (low word) and keyboard control code (high word).
SV_ENDDRAGKB	85	Keyboard message (low word) and keyboard control code (high word).
SV_SELECTKB	86	Keyboard message (low word) and keyboard control code (high word).
SV_OPENKB	87	Keyboard message (low word) and keyboard control code (high word).
SV_CONTEXTMENUKB	88	Keyboard message (low word) and keyboard control code (high word).

*(continued)*



Table 4.5 (Continued)

<i>Define</i>	<i>Value</i>	<i>Description</i>
SV_CONTEXTHELPKB	89	Keyboard message (low word) and keyboard control code (high word).
SV_TEXTEDITKB	90	Keyboard message (low word) and keyboard control code (high word).
SV_BEGINSELECTKB	91	Keyboard message (low word) and keyboard control code (high word).
SV_ENDSELECTKB	92	Keyboard message (low word) and keyboard control code (high word).
SV_ANIMATION	93	Window animation on (TRUE) or off (FALSE).
SV_ANIMATIONSPEED	94	Window animation speed in milliseconds.
SV_MONOICONS	95	Black-and-white icons.
SV_KBDALTERED	96	Hardware ID of new keyboard.
SV_PRINTSCREEN	97	Print Screen enabled (TRUE), disabled (FALSE).
SV_CSYSVALUES	98	Number of system values.

To size a window that is already displayed on the screen, the simplest approach is via the *WinSetWindowPos()* and *WinSetMultWindowPos()* functions. This last function acts on several windows simultaneously. With *WinSetWindowPos()* you just have to specify new values for the *cx* and *cy* parameters, together with the *SWP\_SIZE* flag. Naturally, the handle you must provide has to refer to the frame window that will in turn take care of repositioning all of its control windows, according to its own new position. To find the position of a window on the screen, the *WinQueryWindowPos()* function is handy:

```
#define INCL_WINMESSAGEGR
BOOL WINAPI WinQueryWindowPos( HWND hwnd, PSWP pswp );
```

<i>Parameter</i>	<i>Description</i>
<i>hwnd</i>	The window's handle
<i>pswp</i>	Address of a SWP structure
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

This function fills an SWP structure with data indicating the on-screen position of the window referred to by the *hwnd* handle; all resulting values are expressed in the coordinates of the parent window. To know the exact location of a window in PM in terms of desktop coordinates, it is enough to indicate the frame window's handle rather than the client window's handle. This function stores all information in an SWP structure like this:

```

typedef struct _SWP
{ // swp
  ULONG f1 ;
  LONG cy ;
  LONG cx ;
  LONG y ;
  LONG x ;
  HWND hwndInsertBehind ;
  HWND hwnd ;
  ULONG u1Reserved1 ;
  ULONG u1Reserved2 ;
} SWP ;

typedef SWP *PSWP ;

```

The `f1` member contains one or more flags regarding the window's on screen position. These flags are summarized in Table 4.6.

If the position of a client window is being asked for, then the values returned in the `x` and `y` members will be, respectively, the width and height of the sizing border created with the `FCF_SIZEBORDER` flag, as defined by system level settings. Remember that a window position and size area are always expressed in terms of its parent. For a client it is its frame window. The lower-left corner of a client lies only a few pixels from the same frame window's corner at exactly the border dimensions.

**Table 4.6 Styles Contained in an *fl* Member of a SWP Structure**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
SWP_SIZE	0x0001	Changes the window's size.
SWP_MOVE	0x0002	Moves the window.
SWP_ZORDER	0x0004	Changes the window order in the windows management list.
SWP_SHOW	0x0008	Displays the window.
SWP_HIDE	0x0010	Hides the window.
SWP_NOREDRAW	0x0020	Prevents the window from being redrawn.
SWP_NOADJUST	0x0040	Avoids the issuing of the WM_ADJUSTWINDOWPOS message.
SWP_ACTIVATE	0x0080	Activates the window.
SWP_DEACTIVATE	0x0100	Deactivates the window.
SWP_MINIMIZE	0x0400	Minimizes the window.
SWP_MAXIMIZE	0x0800	Maximizes the window.
SWP_RESTORE	0x1000	Restores the window's position and size that it had prior to maximize or minimize operation.
SWP_FOCUSACTIVATE	0x2000	Gives the focus to the window.
SWP_FOCUSDEACTIVATE	0x4000	Takes away the focus from the window.

If, instead, you specify the handle of a frame window, then the values returned in *x* and *y* are expressed in screen coordinates, and therefore indicate the exact position of the window with respect to the desktop.

The data contained in each single window's reserved memory area are useful for determining the restore position and the minimize position of a generic PM window. Furthermore, you can call the *WinGetMinPosition()* function to find the minimize position of a window.

---

## Saving the Position of a Window

It is far more interesting to examine some new functions implemented in OS/2 2.1 for managing windows on the screen. These functions are *WinStoreWindowPos()* and *WinRestoreWindowPos()*.

Any OS/2 2.1 user must have noticed that the system is able to restore windows on the screen exactly in the position they held when the system was last shut down. This behavior, which can be disabled with a specific directive in CONFIG.SYS, is very effective and convenient because it allows the user to store a great deal of information regarding a window simply by resorting to a single API function. The storage of information takes place in OS2.INI, a system initialization file that acts as a true information repository for all active OS/2 applications. What is surprising about the *WinStoreWindowPos()* and *WinRestoreWindowPos()* functions is their plain syntax. They only need three parameters, which are self-explanatory.

```
#define INCL_WINWORKPLACE
BOOL APIENTRY WinStoreWindowPos(    PSZ pszAppName,
                                     PSZ pszKeyName,
                                     HWND hwnd) ;
```

<i>Parameter</i>	<i>Description</i>
pszAppName	Application name
pszKeyName	Assigned key
hwnd	Window handle
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

```
#define INCL_WINWORKPLACE
BOOL APIENTRY WinRestoreWindowPos(  PSZ pszAppName,
                                     PSZ pszKeyName,
                                     HWND hwnd) ;
```

<i>Parameter</i>	<i>Description</i>
pszAppName	Application name
pszKeyName	Assigned key
hwnd	Window handle
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operations

Starting from the end of the parameter list, the last parameter identifies the window that needs to be a snapshot of all its attributes (position and size, but also other presentation parameters). In general, this will be a frame window. To store the information in the INI file, it is necessary to define a pair of text strings corresponding to the application's name and to a distinctive key. For instance, imagine you are dealing with an application called TWENY.EXE. One correct pair of strings would be TWENY and POSITION, but also TOKIO and JAPAN. It is important to permanently store these strings in a safe place inside the application. The return value is a boolean that reports the success or the failure of the operation.

Using these two functions ensures that all windows featured by a program will be redisplayed exactly at their previous position, with the same size and attributes (background color, text color, and text font). These functions are easy to use in your own applications, or to plan for some option in the user interface (a menu item) that might allow the user to disable this functionality. Let's use the pair of functions on a single application window. The logic to be implemented should create a window without the FCF\_SHELLPOSITION flag. Immediately after the window's creation, you can call `WinRestoreWindowPos()` in order to retrieve information from OS2.INI. If this is the first-ever execution, then the function will return the value of FALSE, so it will be necessary to take care of the window's positioning with `WinSetWindowPos()`. The storage of the window data in the initialization file takes place after the message loop, and before the application's termination. In Listing 4.4 you can also see the usage of the `WinQueryTaskSizePos()` function, which locates an optimal position on the screen for the application's main window.



Let's now examine what is involved in the development of applications that display several windows simultaneously, by putting into practical use the concepts you have learned about parenthood and ownership.

---

## Creating a Client's Child Window

Starting with version 1.2 of OS/2, the system editor is a full-blown PM application. Actually, it is a simple text editor, with a top-level window, a sizing border, and some basic functionalities guaranteed by some menu options (it really is a 16-bit application that has been ported to the 32-bit world without enhancing its look or overall functionality).

E.EXE is made up of a top-level window which displays a WC\_MLE class window precisely overlapping the client window. Remember that a WC\_MLE window is a *multiline entry field*. Thus the client is never visible, because it is hidden behind the WC\_MLE window. Its existence, however, can be proven through intelligence tools, by assessing the class to which the WC\_MLE window's parent belongs. You will discover that it is in the EHXMAIN class registered by the editor. The sizing of the top-level window, by acting on its border, causes an equal variation in the mle window; the net result is that it will cover the whole screen area occupied by the editor. This mechanism is performed by the client window and by the other frame controls as a consequence

of receiving appropriate sizing messages issued by the frame window. There is no way in which the *mle* window can be hooked into the logic of the automatic resizing, which governs the frame and its controls. Therefore, it is necessary that the application itself takes care of implementing this behavior by intercepting the appropriate message.

In reality, it is possible to trick PM by indicating in place of the client window any other window belonging to a predefined class. This method allows you to avoid registering a window class and writing a window procedure. However, it does require *subclassing* of the *mle* window in order to track the message flow generated by the user's interacting with menus. As we have seen, the creation of a window deprived of a client window can be done through *WinCreateWindow()* simply by specifying the `WC_FRAME` class. Once a `WC_MLE` class window has been created, it is sufficient to enforce it to have the frame window as its parent, and assign it an ID with the value of `FID_CLIENT`. It is simpler to use the *WinCreateStdWindow()* function that requires you to indicate the `WC_MLE` class directly as the one to which the client window belongs. In Chapter 7, where the predefined window classes will be described, this issue will be explained in detail.

Despite all the previous suggestions, it is essential to find another solution to this problem so that the *mle* window becomes completely bound to the application's frame window (even though the system does not force this automatically). In Figure 4.9 you can see two top-level identical windows each containing a `WC_MLE` class window. Consider this as a first attempt to build an editor application for PM. The different dimensions of the two windows are indicative of the fact that one of them must have been resized; also, they prove that the child windows have been affected by the resize operation.

While developing this application, you will become aware of two essential principles: the initial size of the input window (the *mle* window) conforms to the client window and to any subsequent rearrangement of the main window. The `WC_MLE` class window is served by a proper window procedure inside PM, as this is one of the predefined classes. The message flow addressed to the *mle* window, is therefore inaccessible to the application. When the main window is created through *WinCreateStdWindow()*, the message `WM_CREATE` is passed to the application's window procedure. Catching this message provides an ideal opportunity for creating the `WC_MLE` window, exploiting *WinCreateWindow()*. This time the message `WM_CREATE` is issued and addressed to the proper window procedure inside PM. Later, in the application's window procedure, the message `WM_SIZE` is issued as a direct consequence of the main window's creation. (We will discuss in detail the entire message flow produced by creating a window in Chapter 7.) The contents of `mp2` correspond to the dimensions of the client window of the main window. By extracting from `mp2` the two `SHORT` values representing, respectively, the width and the height of the client window, it is possible to resize the input window as the actual size of the client is known.

This approach to the problem has two advantages: It allows a `WC_MLE` class window to be resized either when the application is created or in response to any subsequent

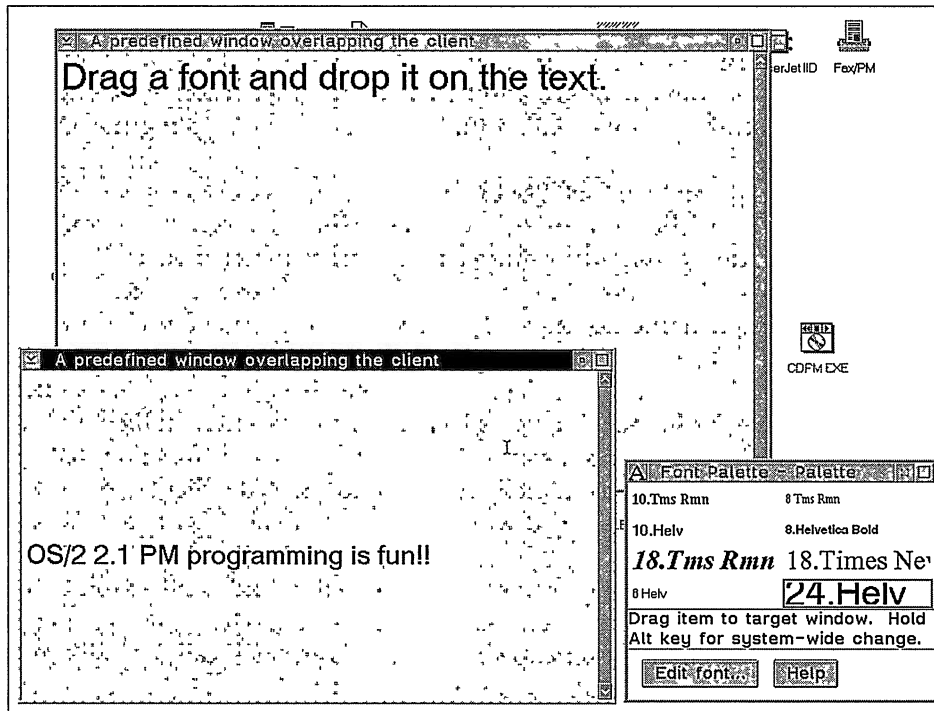


Figure 4.9 Two top-level windows, each one containing a WC\_MLE class window.

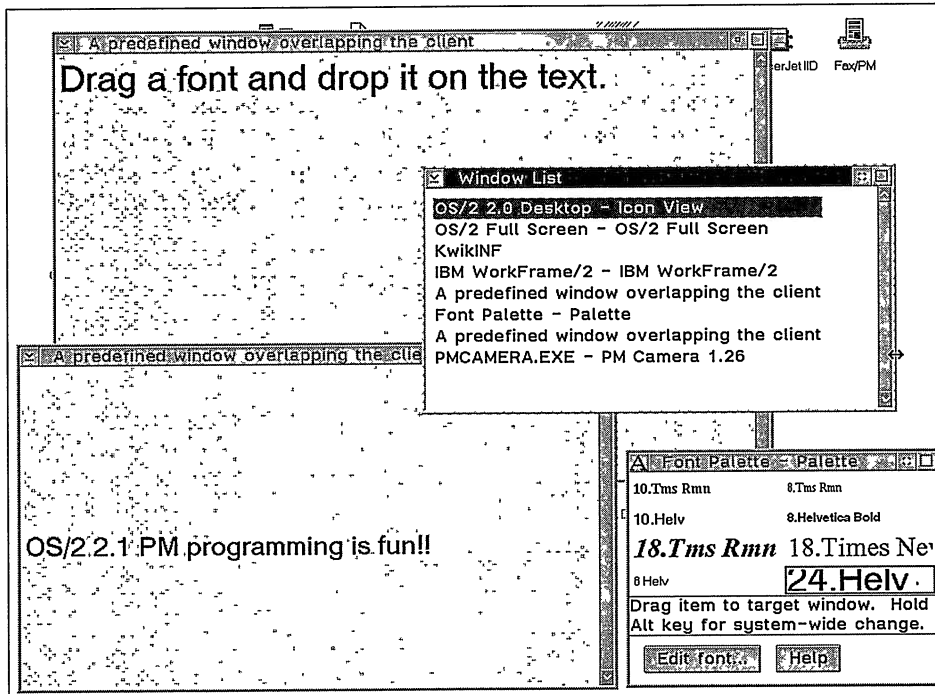
changes made by the user to the window's borders. In fact, as the window's class has been registered with the CS\_SIZEREDRAW flag set, it is guaranteed that a WM\_SIZE message will be issued for any kind of change along either axis.

The only operation involved is that of catching the WM\_SIZE message and passing the value contained in mp2 to the *WinSetWindowPos()* function, specifying the handle of the input window. The finishing touch will be that of declaring static storage class for the handle of the WC\_MLE class window. This data item is returned within the block of code dealing with the WM\_CREATE message, and it is used once again with the WM\_SIZE message—two very distinct moments at which the window procedure is executed. Listing 4.5 contains only the window procedure for the example at hand.



## Informing the Window List

When a PM application is running, Window List is informed automatically about its presence. In this way it can give the user a complete overview of what tasks are active at any given moment (see Figure 4.10). PM applications must have the FCF\_TASKLIST flag set in order to reproduce in the Window List the titlebar's text.



**Figure 4.10** The Window List displays a list of active and inactive PM applications at any given moment.

Quite often, however, it is preferable to notify Window List with a text string invented specifically for this purpose; this is what most OS/2 applications provided by the system actually do. The function used is `WinAddSwitchEntry()` which can pass to Window List the address of a `SWCNTRL` structure. However, before doing this, it is necessary to have some other items, like the process's PID and TID. These acronyms stand for, respectively, the *process identification number* and the *thread identification number*. These are two `ULONG` numbers that identify the process and the thread being executed. The first value is unique at the system level (each process, whatever kind of program it might be, is assigned a unique PID). The second number is also unique, but at the process level, and always starts from the value of one (primary thread). You can obtain these two values with the `WinQueryWindowProcess()` function:

```
#define INCL_WINMESSAGEMGR
BOOL APIENTRY WinQueryWindowProcess(  HWND hwnd,
                                       PPID ppid,
                                       PTID ptid) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Window handle
ppid	Pointer to an identifier of type PID
ptid	Pointer to an identifier of type TID

<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

This function reports the PID and TID of the current process and thread to which the window indicated by the first parameter belongs (in our example, it would be the primary thread). Once this has been accomplished, you can supply the correct values to the SWCNTRL structure that you will eventually pass to the *WinAddSwitchEntry()* function:

```
#define INCL_WINSWITCHLIST
HSWITCH APIENTRY WinAddSwitchEntry( PSWCNTRL pswcntrl ) ;
```

<i>Parameter</i>	<i>Description</i>
swcntrl	Address of a SWCNTRL structure

<i>Return Value</i>	<i>Description</i>
HSWITCH	Handle of the new string inserted into the Window List

```
typedef struct _SWCNTRL
{
    // swctl
    HWND hwnd ;
    HWND hwndIcon ;
    HPROGRAM hprog ;
    PID idProcess ;
    ULONG idSession ;
    ULONG uchVisibility ;
    ULONG fbJump ;
    CHAR szSwtitle[ MAXNAMEL + 4 ] ;
    ULONG bProgType ;
} SWCNTRL ;

typedef SWCNTRL *PSWCNTRL ;
```

The first member identifies the handle of a top-level window's frame window; the next member is the handle of the icon associated with that window. The program's handle is an item known by calling the *PrfQueryProgramHandle()* function, but it can also be omitted from the *WinAddSwitchEntry()* call.

The process ID corresponds to the PID value you retrieved previously, while the session ID is returned by the system at start-up time, through *DosStartSession()*. The *uchVisibility* member defines the application's visibility in Window List; the values it can take are: SWL\_GRAYED, SWL\_INVISIBLE, and SWL\_VISIBLE.

<i>Flag</i>	<i>Value</i>	<i>Description</i>
SWL_INVISIBLE	0x01	The text string is invisible
SWL_NOTJUMPABLE	0x01	You cannot access to the process through its corresponding string
SWL_GRAYED	0x02	The text string is visible, but it cannot be selected
SWL_JUMPABLE	0x02	You can access to the process through its corresponding string
SWL_VISIBLE	0x04	The text string is visible



Through fbJump you define whether the application is part of the jump selection sequence. The structure is then completed with the title inserted in the Window list and a reserved parameter that always takes the value of zero.

Figures 4.11 and 4.12 show Window List before and after the addition of the code statements that insert text in Window List. In Figure 4.11 you can also see the name of the executable module, while in Figure 4.12 it has been replaced by the string "Boca Raton, FL!"

## Execution of One Single Instance

OS/2 2.1 is a multitasking environment that allows simultaneous execution of several programs at once. Nothing forbids that among the many active applications you could have several copies of the *same* program; in other words, you could have multiple instances of the same EXE. By default, the WPS interface does not allow more than one single copy of any program to be loaded, starting from its icon that refers to its executable module. This can be altered by changing some of the object's properties on the *Window* page in the Settings notebook. The standard behavior can also be modified on the Window object in the System object in the *System Setup* folder. When an OS/2

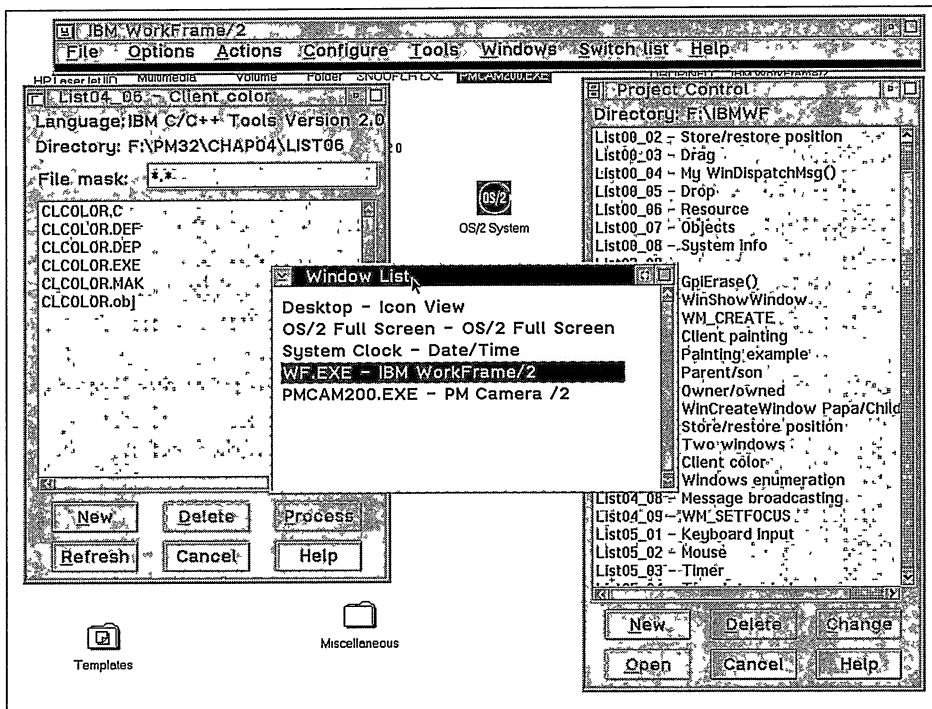
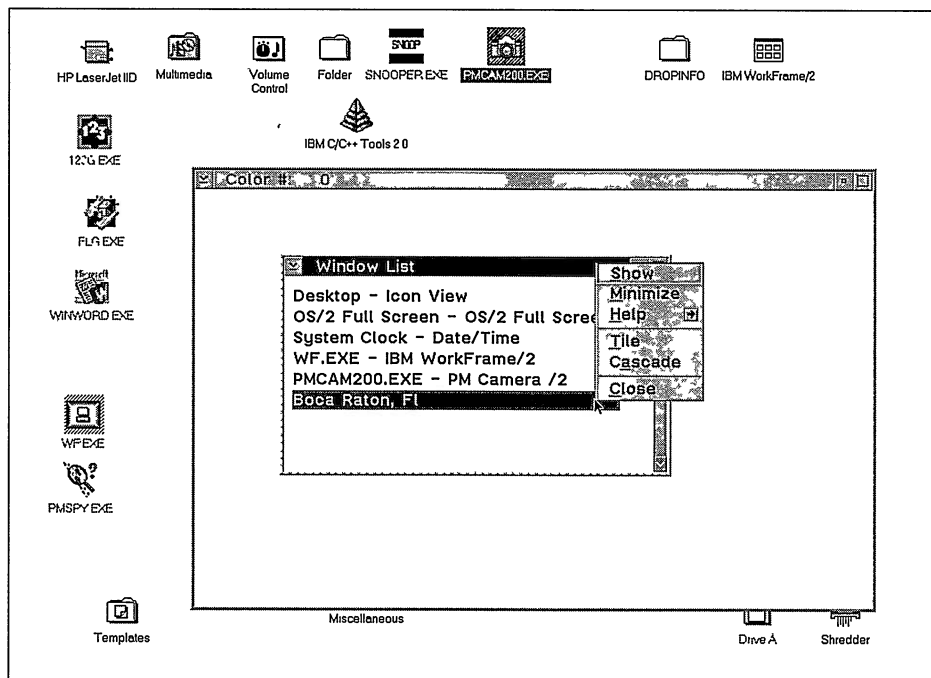


Figure 4.11 How to act on the contents of the Window List through the *WinAddSwitchEntry()* function.



**Figure 4.12** Change of the contents of the Window List thanks to the *WinAddSwitchEntry()* function.

application is run with multiple instances, the code segment is shared among them, while there will be distinct data segments for each single instance.

The execution of just a single instance of a Windows program in the MS Windows environment is a very simple operation—and a very common one. In this case it is the system itself that informs the application about any other possible preceding instances.

In PM, on the other hand, things are not this simple. The *main()* function is the entry point of any application, and it is not passed any special information from the system. The only possible parameters it can receive are *argc* and *\*argv[]*, and perhaps *\*envp[]*. Furthermore, there is no special item that indicates whether a preceding instance is already running or not. Each instance is an independent process with private resources like PM window classes. This means that they are inaccessible and invisible from the outside of the module within which they have been registered. Therefore it is necessary to find some alternative means that exploit the Window List notification process of the existence of an application or the enumeration of the top-level windows present in the screen group.

## *Accessing the Window List*

With this approach, you must have a basic piece of information, and that is the text that the application, once it has been started, passes over to Window List. This text

string should be one of the many resources that make up an application, and that are present in the RC file. As we have not yet examined this aspect of PM programming, right now we will simply hard-code the text that is used in Window List inside a character array. The information will be used to make comparisons between the items contained in Window List and the application being executed. The presence of an equal string in Window List is clear evidence that a preceding instance of the application is already running (and therefore that you will have to assign some new specific text string in order to avoid this algorithm being invalidated).

The first operation is that of obtaining the items that are present in Window List's listbox. The *WinQuerySwitchList()* is adequate for this purpose:

```
#define INCL_WINSWITCHLIST
ULONG APIENTRY WinQuerySwitchList(HAB hab,
                                   PSWBLOCK pswblkSwitchEntries,
                                   ULONG ulDataLength) ;
```

<i>Parameter</i>	<i>Description</i>
hab	Anchor block handle
pswblkSwitchEntries	Address of an identifier of type SWBLOCK
ulDataLength	Size of the data being passed
<i>Return Value</i>	<i>Description</i>
ULONG	Total number of switch entries in the Window List

The first parameter is the anchor block handle of the application; if you need it you can retrieve it through *WinQueryAnchorBlock()*. The second parameter is a pointer to a SWBLOCK structure, as follows:

```
typedef struct _SWBLOCK
{
    // swblk
    ULONG cswentry ;
    SWENTRY aswentry[ 1 ] ;
} SWBLOCK;

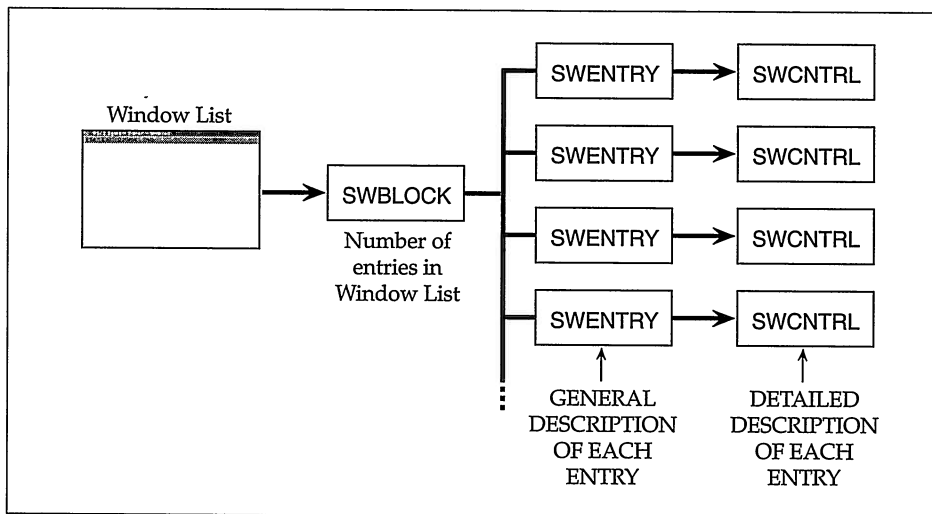
typedef SWBLOCK *PSWBLOCK;
```

The first member of the SWBLOCK structure indicates the number of items contained in the following array of SWENTRY items. Every element in this array is thus a SWENTRY structure:

```
typedef struct _SWENTRY
{
    // swent
    HSWITCH hswitch ;
    SWCNTRL swctl ;
} SWENTRY ;

typedef SWENTRY *PSWENTRY ;
```

The SWBLOCK structure contains the number of entries present in Window List and described by the array of SWENTRY structures. In each SWENTRY structure you have each entry's handle (i.e., the application handle), as it is managed by Window List, and a



**Figure 4.13** Relationship between the Window List and the active processes in the system.

detailed description of the program contained in a SWCNTRL structure. Figure 4.13 summarizes the relationship existing between Window List, each entry that appears in its listbox, and the data structures checked by *WinQuerySwitchList()*.

The first operation to perform is that of determining the number of entries in the Window List's listbox by specifying NULL and 0L in place of the second and the third parameter of *WinQuerySwitchList()*:

```

...
ulApps = WinQuerySwitchList( hab, NULL, 0L );
...

```

The *ulApps* identifier is a ULONG value that represents the number of entries in Window List; this information will be used to allocate a block of memory large enough to contain a corresponding SWBLOCK structure.

```

...
ulSize = ulApps * sizeof( SWENTRY ) + sizeof( USHORT );
DosAllocMem( &pswblk, ulSize, PAG_READ | PAG_WRITE | PAG_COMMIT );
...

```

The *ulSize* identifier, which is also a ULONG value, is equal to the size of an SWBLOCK structure featuring an array of *ulApps* SWENTRY structures. Once the memory block has been allocated with the PAG\_COMMIT flag, you will automatically have a pointer to the correct SWBLOCK structure. Then you only have to call *WinQuerySwitchList()* again to fill in the allocated memory block with the data retrieved from Window List.

```

...
WinQuerySwitchList( hab, pswblk, ulSize );
...

```

The search for any existing instance at run time implies that you need to examine every single entry in the Window List's listbox, and compare it against the text string of your application (`szWindowTitle`): A positive match means that there is a previous instance. The upper limit of this loop is given by the `cswentry` member in the `SWBLOCK` structure.

```

...
for( i = 0; i pswblk -> cswentry; i++)
{
    if( strcmp( pswblk -> aswentry[ i].swctl.szSwtitle,
               ,          szWindowTitle) == 0)
    {
        // match found: terminate application
        return FALSE ;
    }
}
// match not found: continue application
...

```

The piece of code described in this paragraph lies inside *main()* and can be utilized at the beginning of the function before performing any other program specific operation. The only thing to be careful about is that you must have readily available the text string that is to be compared against Window List entries, or simply the name of the program if no customization has been performed with *WinAddSwitchEntry()*. Listing 4.6 shows how to check for an existing instance by examining the contents of Window List.



## Enumerating Top-Level Windows

The second solution is discovering all of the desktop's child windows, and then examining each one of them. You then compare them against the window that is about to become the application's main window. As opposed to the preceding solution, you will now have more items to compare (the application's title and the name of the class to which it belongs), and the implementation is somewhat simpler. As far as performance issues are concerned, it is difficult to decide which is best, since the algorithm is very simple in both cases. This second solution is certainly viable, if not only for the purpose of enumerating all child windows (of any given window, not necessarily the desktop).

The algorithm for enumerating all PM windows is surprisingly simple. The functions involved are only three: *WinBeginEnumWindows()*, *WinGetNextWindow()*, and *WinEndEnumWindows()*. The roles played by the three functions are distinct and self-explanatory. Start by counting the number of windows with *WinBeginEnumWindows()*:

```

#define INCL_WINWINDOWMGR
HENUM APIENTRY WinBeginEnumWindows( HWND hwnd) ;

```

<i>Parameter</i>	<i>Description</i>
hwnd	Window handle

<i>Return Value</i>	<i>Description</i>
HENUM	Enumeration handle

This function takes as its sole parameter the handle of the window whose children you need to find; it can be any HWND handle or a HWND\_DESKTOP and HWND\_OBJECT define that indicate, respectively, the desktop window and the object window of the PM screen group. The return value is an enumeration handle, a special tool for this kind of operation. An enumeration handle is used with the *WinGetNextWindow()* and *WinEndEnumWindows()* functions, to get the first child window of the window indicated in the call to *WinBeginEnumWindows()*, and to terminate the search operation, respectively. The entire operation is performed through the enumeration handle, which is managed by the system. *WinGetNextWindow()* returns the handle of the next window found, or a NULLHANDLE if the enumeration has reached its end:

```
#define INCL_WINWINDOWMGR
HWND APIENTRY WinGetNextWindow( HENUM henum ) ;
```

<i>Parameter</i>	<i>Description</i>
henum	Enumeration handle

<i>Return Value</i>	<i>Description</i>
HWND	Window handle or NULLHANDLE at the end of the search

Whatever window is indicated by the hwnd handle returned by *WinGetNextWindow()*, it will always refer to a window of the WC\_FRAME class, provided the standard PM approach to window construction has been followed. Once you have a window's handle, you find all information that qualifies it and sets it apart from all other windows. If you wish to be sure that it really is WC\_FRAME class window, just call the *WinQueryClassName()* function:

```
#define INCL_WINWINDOWMGR
LONG APIENTRY WinQueryClassName( HWND hwnd,
                                LONG cchMax,
                                PCH pch ) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Window handle
cchMax	Buffer length
pch	Buffer for storing the class name

<i>Return Value</i>	<i>Description</i>
LONG	Number of characters inserted into the buffer

This function writes into the pch buffer—the third parameter—the name of the class of the window referenced by the hwnd handle, the first parameter. The cchMax long value indicates the total length of the buffer, and *WinQueryClassName()* copies into the PCH buffer the first cchMax - 1 characters of the class's name.

For any window belonging to one of the fifteen predefined PM windows, a specific alphanumeric string is stored in the buffer. This special string has # as its first character, then a number. Table 4.7 lists all values corresponding to the predefined window classes.

**Table 4.7 The Predefined PM Window Classes and the Corresponding Values Returned by *WinQueryClassName()***

<i>Class</i>	<i>Name</i>
WC_FRAME	((PSZ)0xffff0001L)
WC_COMBOBOX	((PSZ)0xffff0002L)
WC_BUTTON	((PSZ)0xffff0003L)
WC_MENU	((PSZ)0xffff0004L)
WC_STATIC	((PSZ)0xffff0005L)
WC_ENTRYFIELD	((PSZ)0xffff0006L)
WC_LISTBOX	((PSZ)0xffff0007L)
WC_SCROLLBAR	((PSZ)0xffff0008L)
WC_TITLEBAR	((PSZ)0xffff0009L)
WC_MLE	((PSZ)0xffff000AL)
WC_SPINBUTTON	((PSZ)0xffff0020L)
WC_CONTAINER	((PSZ)0xffff0025L)
WC_SLIDER	((PSZ)0xffff0026L)
WC_VALUESET	((PSZ)0xffff0027L)
WC_NOTEBOOK	((PSZ)0xffff0028L)

Once you have found that the window is indeed a WC\_FRAME class window (#1), you can then get the application's title with *WinQueryWindowText()*. The operation is performed automatically by this function, which stores in a buffer whatever appears in the titlebar. Now you only need to compare the retrieved string against that known by the application, and determine if a previous instance is already running. The example is shown in Listing 4.7.



### ***A Third Solution***

The problem of executing only one single instance of an application can be solved in several ways in PM's set of API functions. Let's examine a third solution based on the unique nature of the inner workings of the whole environment: message passing. PM's API includes several hundred predefined messages, among which the WM\_ messages are predominant and are the only ones that have been considered so far. However, there are also other categories of messages, like the WM\_USER messages that are defined and managed entirely by the application itself. A WM\_USER message takes on the value of 0x1000 and represents the starting point for defining new messages, in the form of WM\_USER + n, where n is any positive integer.

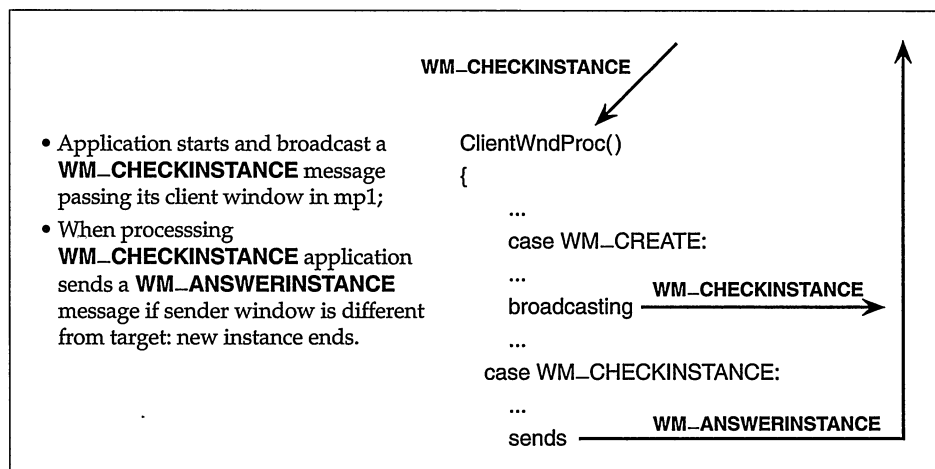
In addition to defining the message's name, the application can freely utilize the 64-bit space available in the mp1 and mp2 parameters. Immediately after being activated, an application that needs to be running as the first and sole instance can issue a customized message to all other active applications (windows). The message defined

in the code can be intercepted and processed only by another running instance of the same application. The response implemented by the code receiving the message issues another different customized message to the new instance. This second message simply causes the application to set a file scope boolean value to FALSE. The application's message loop and the *WinShowWindow()* function that displays the main window are inside a block of code based on the value of the boolean identifier. This solution is determined by the requirement of not having to show the new window, not even for a fraction of a second. If instead the instance's response would have been its *posting* of the WM\_QUIT message, then you would not achieve the desired result. Figure 4.14 shows the logic that governs this solution.

The issuing of a message to all top-level windows present in the PM screen group can be accomplished through the *WinBroadcastMsg()* message:

```
#define INCL_WINMESSAGEMGR
BOOL APIENTRY WinBroadcastMsg(HWND hwnd,
                               ULONG msg,
                               MPARAM mp1,
                               MPARAM mp2,
                               ULONG rcf) ;
```

*WinBroadcastMsg()* is a function with a convoluted rationale, but it is easy to use. The first four parameters are exactly those of *WinSendMsg()* and *WinPostMsg()*—the parameters required by any generic window procedure. However, the fifth argument is an unsigned long value that instructs *WinBroadcastMsg()* as to how it should broadcast a message. In fact, this function allows a message to be sent directly to a window procedure by means of the message queue or to post the message to one or more applications. To decide which of the two ways you should use, set one of the following flags as the function's fifth parameter:



**Figure 4.14** Functional scheme of the execution of a single application instance through the posting of a customized message.



<i>Flag</i>	<i>Value</i>	<i>Description</i>
BMSG_POST	0x0000	The message is posted
BMSG_SEND	0x0001	The message is sent
BMSG_POSTQUEUE	0x0002	The message is addressed only to the applications' queues
BMSG_DESCENDANTS	0x0004	The message is addressed to all child windows as well
BMSG_FRAMEONLY	0x0008	The message is addressed to WC_FRAME class windows only

With BMSG\_POST, the message is posted to all child windows of the window indicated by the *hwnd* parameter, the first parameter of *WinBroadcastMsg()*. With BMSG\_POSTQUEUE, on the other hand, you instruct *WinBroadcastMsg()* to deliver the message to all system threads that have a message queue. In this case, the value of the target window is set to NULLHANDLE. Finally, with BMSG\_SEND you can send a message to all window procedures of the child windows of the window indicated by the first parameter of *WinBroadcastMsg()*.

There are yet two more flags that can differentiate the type of windows that will receive the message issued by *WinBroadcastMsg()*. BMSG\_DESCENDANTS indicates that all child windows should receive the message, and BMSG\_FRAMEONLY indicates that only the windows belonging to the WC\_FRAME class should receive the message. You can thus decide how to set up the *WinBroadcastMsg()* function for your own needs.

Naturally, you do not know beforehand what the handle of the receiver window is; rather, this is what you are trying to find out. Therefore, the first parameter of *WinBroadcastMsg()* will necessarily have to be HWND\_DESKTOP to indicate the forefather of all windows in the system. The message that will be broadcast to detect any possible presence of previous instances is WM\_CHECKINSTANCE, defined as follows in the application:

```
...
#define WM_CHECKINSTANCE WM_USER + 0
...
```

The WM\_CHECKINSTANCE message is just an alias for WM\_USER with no increment. In addition to the message itself, you also need to provide a piece of information that will be critical if another running instance is to be detected: the handle for the newly created window. This parameter is used by a possible existing previous instance to force the closure of the new instance. The handle of the window is packed into *mp1* by means of MPFROMHWND( *hwnd* ), while all 32 bits of *mp2* are zeroed out by setting it to 0L. The options are BMSG\_FRAMEONLY and BMSG\_SEND. The whole thus becomes:

```
...
WinBroadcastMsg( HWND_DESKTOP, WM_CHECKINSTANCE, MPFROMLONG( hwnd ),
0L, BMSG_FRAMEONLY | BMSG_SEND ) ;
...
```

The choice of BMSG\_SEND is absolutely critical to terminate execution of any current instance. The broadcasting of a message through *WinBroadcastMsg()* implies that you

will reach *all* window procedures of the client window (the frame window will pass on the message to its client directly). Before returning to `WinBroadcastMsg()`, the message will probably have reached even a window procedure of some existing instance of the application. This window procedure is implemented in order to provide a case condition for `WM_CHECKINSTANCE`. In this portion of the window procedure's code two fundamental operations are performed. The first is assessing that the message issued by `WinBroadcastMsg()` has not and will not reach the window procedure of the sender (as it is the same piece of code, even the new instance will share it). Once it has been determined that the handle contained in `mp1` is different from the handle of the current window procedure's client window, the new instance must be closed.

To solve this problem, there is no other way than posting the `WM_QUIT` message into the message queue of the second instance. Actually, this second solution is not *entirely* correct, because it assumes you are using `WinPostMsg()` to implement any kind of communication between separate applications, in addition to allowing that the new window will be displayed for a short time, and it is during that time that it returns from the message loop and retrieves the `WM_QUIT` message.

It is possible to solve the problem with a new `WinPostMsg()`, this time specifying the destination window's handle (the information is contained in `mp1`), the `WM_ANSWER-INSTANCE` message, and nothing in `mp1` and `mp2`.



In practice, this solution can be reduced to just a few lines of code, and to the use of the `WinBroadcastMsg()` API function. Listing 4.8 shows the source code calling `WinBroadcastMsg()` to solve the problem of a running a single application instance.

## Other Solutions

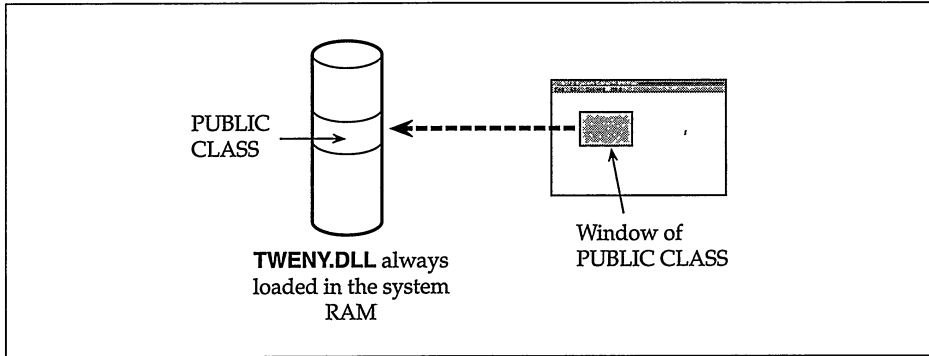
The problem of running a single application instance can be defined by other means as well, all provided by the system. One very simple means is that of creating and setting a shared runtime application *semaphore*. Each program invocation will check the semaphore's status before actually creating the instance. The first instance will get an error status as a return value. Any subsequent instances will terminate as soon as they have found the semaphore, and, thus, a previous instance.

Another way is to create a *named pipe* and connect to it as a client. This can be done with only one application. All others will fail and thus can be terminated. Probably, with some imagination, you can devise other tricks that do the job.

---

## Registering a Public Class

In PM, all classes registered in a single application are private. A private class of windows means that only the current process is allowed to create windows belonging to it. This is true even for the execution of multiple instances of the sample application; every instance takes care of registering the classes implemented in their source code, and of managing the information related to these classes on a strictly local basis.



**Figure 4.15** If a window class is to be public, then it has to be registered directly in a DLL.

In order to create a public class, you need to set the `CS_PUBLIC` flag when the class is registered. A public class also requires that the registrations take place in a dynamic link library (DLL), so that the associated window procedure is accessible to any applications. Now, things start to get complicated! In Figure 4.15 you can see the nature of a public class.

At the current stage we do not have enough elements available to be able to register a public class and then to create windows belonging to that window class. This will be covered when DLLs are explained in Chapter 10.

Furthermore, it is also necessary that the `OS2.INI` system configuration file contain a specific directive to instruct all applications about the presence of any new window class that might be exploited by any piece of runtime code. The software designer can take advantage of the functions with a *Prf* prefix for dealing with all kinds of configuration problems, from reading a file to customizing the system's profile. When we get back to the problem of creating a public window class, we will examine the *Prf* functions in detail. For the moment, just remember to limit any changes to `OS2.INI` to the bare minimum, because if you do anything wrong, you will run into subsequent problems when rebooting the system: In other words, you might get a total crash.

The somewhat greater complexity in creating a public window class in PM is also due to the protected-mode nature of any runtime tasks, and this entails major consequences as far as the whole system's stability is concerned.

---

## The `WinMessageBox()` Function

The best way to determine what is actually happening in a PM application is by using IPMD, possibly in a dual monitor configuration (when supported), because this simplifies the handling of all information.

Quite often, however, it is sufficient to get a "quick and dirty" display of such information, both in order to inspect the code, as well as notifying the user that some

kind of unique condition is established. Instead of creating a special window and filling it with text, it is far easier to use the *WinMessageBox()* API function. This function is designed specifically for this kind of situation. With *WinMessageBox()* you can create a window inside which the application will be able to present some text simply by means of a pointer to some memory area containing a string. You don't have to be concerned about accessing any special presentation space, nor about screen device contexts. *WinMessageBox()* handles these details automatically, so it is an extremely fast and convenient way for producing output on the screen. This tool can also be employed in more involved situations, like prompting the user for information and returning a value that indicates what the user intends to do. The syntax of *WinMessageBox()* is:

```
#define INCL_WINDIALOGS
ULONG APIENTRY WinMessageBox(  HWND hwndParent,
                                HWND hwndOwner,
                                PSZ pszText,
                                PSZ pszCaption,
                                ULONG idWindow,
                                ULONG flStyle) ;
```

<i>Parameter</i>	<i>Description</i>
hwndParent	Handle of the parent window
hwndOwner	Handle of the owner window
pszText	Text displayed in the message box
pszCaption	Title of the message box
idWindow	ID of the message box
flStyle	Style of the message box
<i>Return Value</i>	<i>Description</i>
ULONG	Value of the user's response

In general, a message box is engaged to provide the user with some very short messages pertaining to the system's activity. Some typical examples are: "Insufficient Memory," and "Missing disk in drive A:." This kind of message must immediately capture the user's attention, and thus it is best to place them right in the middle of the screen. For this reason, the handle that indicates the parent of a message window is usually `HWND_DESKTOP`.

On the other hand, the handle that identifies the message box's owner is much more significant. This window is reactivated automatically once the message box disappears from the screen. As a message box is almost always displayed within a case branch of a window procedure, the handle of the owner window is usually the handle of the window that received the last message dispatched by *WinDispatchMessage()*.

The two strings used by *WinMessageBox()* are often referred to as *text-in* and *text-out* in order to distinguish what will appear inside the window and what will appear inside the title of the message box. Both strings must be ASCIIZ strings, and they can contain escape symbols (like newline `\n`), even though it is best to avoid typical escape sequences such as those used by the *printf()* function. If no pointer is specified for the

parameter indicating the message box's text, then the string "Error!" will be displayed automatically; this indicates that originally *WinMessageBox()* was intended for warning of errors or some kind of erratic application behavior.

The possibility of specifying text strings in an easy and straightforward way often deceives PM rookies. A message box can contain at most a couple of text lines. Once this limit is exceeded, it is preferable to resort to some other output tool available in PM. It certainly requires a complicated solution—for example, a WC\_MLE window—but it will allow much more flexibility and control on behalf of the application than a plain message box.

The fifth parameter to *WinMessageBox()* is a ULONG value used to assign the message box a unique identifier. Actually, this ID is never to be used within an application, because its purpose is that of providing the program's help function with a distinctive element that identifies the source of the WM\_HELP message. It is best to indicate a zero for this parameter because the help function is also the last part being built in a PM application, and thus you can safely assume that you are writing the help system only after you have thoroughly debugged the true application code.

The sixth and last parameter of *WinMessageBox()* is a style that allows the message box to include icons or buttons. These give the user an enhanced and productive way to interact with the application. Table 4.8 contains all the values that the flag of *WinMessageBox()* can take, and that originate buttons in the client portion of the window.

**Table 4.8 The MB\_Flags Used to Generate WC\_BUTTON Class Windows in a Message Box**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
MB_OK	0x0000	The message box contains only an OK button.
MB_OKCANCEL	0x0001	The message box contains an OK button and a Cancel button.
MB_RETRYCANCEL	0x0002	The message box contain a Retry button and a Cancel button.
MB_ABORTRETRYIGNORE	0x0003	The message box contains three buttons: Abort, Retry, and Ignore.
MB_YESNO	0x0004	The message box contains a Yes button and a No button.
MB_YESNOCANCEL	0x0005	The message box contains three buttons: Yes, No, and Cancel
MB_CANCEL	0x0006	The message box contains only a Cancel button.
MB_ENTER	0x0007	The message box contains only an Enter button.
MB_ENTERCANCEL	0x0008	The message box contains an Enter button and a Cancel button.

By looking at Table 4.8 you can see that you can get at most three buttons in a message box; also you may not alter or modify this value because of the closed and predefined nature of this kind of window.

Table 4.9 lists the MB\_ flags that allow the assignment of the status of default button to any of the three WC\_BUTTON class windows in a message box.

The presence of one or more icons in a message box serves the purpose of emphasizing the graphical impact of the message you want to give the user. These icons (Table 4.10) are simply bitmaps that can also be used by any other application to play the role of program icon (naturally, this is not an ideal solution, since it is possible to draw new colored icons with the tools available in the OS/2 Development Toolkit).

The flags listed in Table 4.11 deserve some special attention. A message box is a window that is created and managed by PM according to some very particular and specific criteria, different from what we have seen so far (for example, there is no window class registration). The class of a message box is always WC\_FRAME, just as with any other dialog window.

Despite the differences, a message box is always a window, and thus is subject to the same operating and functional rules of the PM screen group. When a message box appears on the screen, it will receive the input focus: all mouse and keyboard input activities will refer to the message box exclusively. This means that any attempt to select a menu in the message box's parent window will be rejected by the system, and signaled with a beep. This is due to the presence of the MB\_APPLMODAL flag (Figure 4.16). The user can always transfer focus to another window, different from the message box's owner window without a problem because generally it will belong to a different process.

On the other hand, when you wish to totally capture the input focus at the PM screen group level, then you need to set the MB\_SYSTEMMODAL flag. It is absolutely essential to make the message box disappear from the screen before trying to perform any other operation. This flag is used less frequently than the other one, mainly as a means for notifying some critical system level events (Figure 4.17).

The return value of *WinMessageBox()* is an unsigned long and is very important when two or more buttons are present. For this type of message box it is absolutely vital to examine the return value in order to allow the application to determine what selection has been performed by the user (Table 4.12).

**Table 4.9 The MB\_ Flags Qualify Which of the Three Buttons Should Be the Message Box's Default Button**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
MB_DEFBUTTON1	0x0000	The first button is the default button (at least that is the automatic setup without specifying the MB_DEFBUTTON1 flag).
MB_DEFBUTTON2	0x0100	The second button is the default button.
MB_DEFBUTTON3	0x0200	The third button is the default button.

**Table 4.10 The MB\_ Flags That Originate Icons in a Message Box**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
MB_NOICON	0x0000	Excludes the presence of an icon.
MB_CUANOTIFICATION	0x0000	Excludes the presence of an icon.
MB_ICONQUESTION	0x0010	The message box contains a question mark icon.
MB_ICONEXCLAMATION	0x0020	The message box contains an exclamation mark icon.
MB_CUAWARNING	0x0020	The message box contains an exclamation mark icon.
MB_ICONASTERISK	0x0030	The message box contains an asterisk icon.
MB_ICONHAND	0x0040	The message box contains a hand icon.
MB_CUACRITICAL	0x0040	The message box contains a hand icon.
MB_QUERY	MB_ICONQUESTION	The message box contains a question mark icon.
MB_WARNING	MB_CUAWARNING	The message box contains an exclamation mark icon.
MB_INFORMATION	MB_ICONASTERISK	The message box contains an asterisk icon.
MB_CRITICAL	MB_CUACRITICAL	The message box contains a hand icon.
MB_ERROR	MB_CRITICAL	The message box contains a hand icon.

**Table 4.11 The MB\_ Flags That Define the Nature of a Message Box with Reference to the Passing of Focus in the Application or in the System**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
MB_APPLMODAL	0x0000	The message box is of the application modal type.
MB_SYSTEMMODAL	0x1000	The message box is of the system modal type.
MB_HELP	0x2000	The message box contains a help button.
MB_MOVEABLE	0x4000	The message box can be moved on the screen and the system menu will contain the three options: Move, Close, and Switch to...

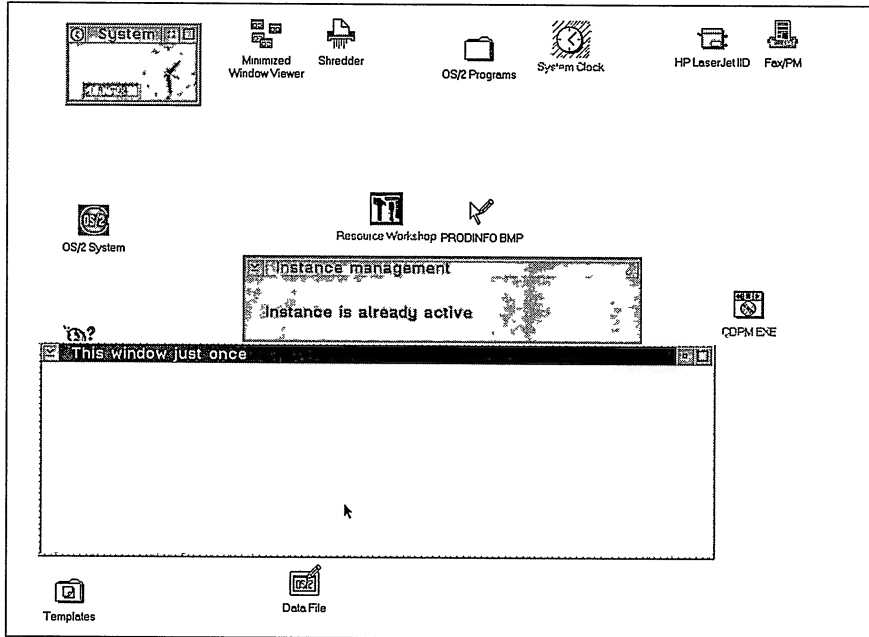


Figure 4.16 If a message box has the MB\_APPLMODAL flag set, then it is possible to transfer focus to another application window.

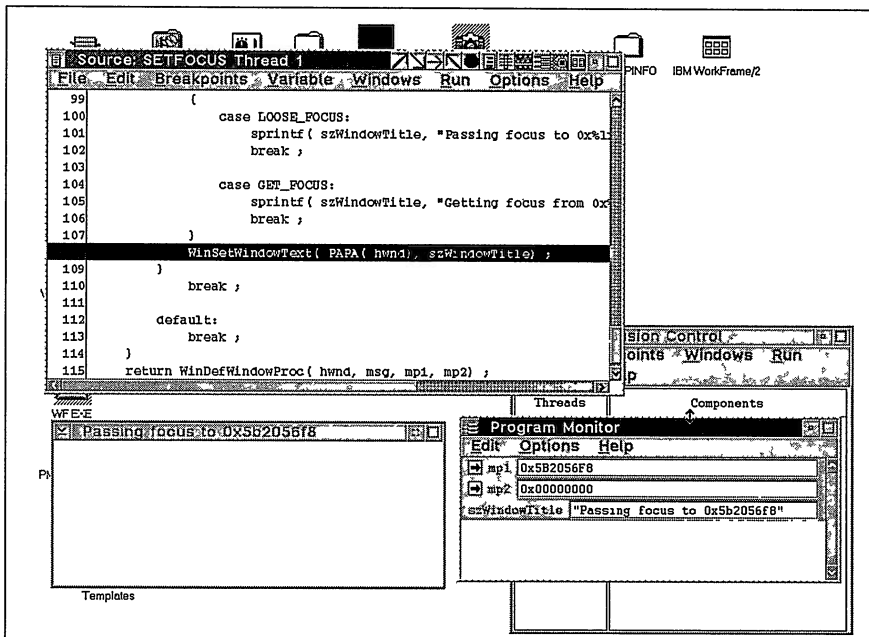


Figure 4.17 A MB\_SYSTEMMODAL message box captures the input focus and will not release it unless it is closed.



**Table 4.12 Return Values Produced by *WinMessageBox()***

<i>Flag</i>	<i>Value</i>	<i>Description</i>
MBID_OK	1	Selection of the OK button
MBID_CANCEL	2	Selection of the Cancel button, or keypress on the ESC key
MBID_ABORT	3	Selection of the Abort button
MBID_RETRY	4	Selection of the Retry button
MBID_IGNORE	5	Selection of the Ignore button
MBID_YES	6	Selection of the Yes button
MBID_NO	7	Selection of the No button
MBID_HELP	8	Selection of the Help button
MBID_ENTER	9	Selection of the Enter button
MBID_ERROR	0xffff	Error in creating the message box

In the following code fragment you can see a sample of *WinMessageBox()*:

```

...
if( !WinLoadString(   hab, NULLHANDLE,
                     ST_MESSAGE,
                     sizeof( szMessage), szMessage))
    return 0L ;

if( !WinLoadString(   hab, NULLHANDLE,
                     ST_TITLE,
                     sizeof( szTitle), szTitle))
    return 0L :

WinMessageBox(       HWND_DESKTOP, hwnd,
                   szMessage, szTitle,
                   0L,
                   MB_OK | MB_ICONASTERISK) ;

...

```

In this example, only one button is present, and thus it is not necessary to catch and decipher the return value. As usual, to specify more flags you can use C's bitwise OR operator.

---

## The Focus Chain

The flags in Table 4.11 introduce an interesting and important subject for developing full-blown PM applications: *focus handling*. This term indicates the set of operations that the system undertakes when the user has selected a window different from the current one; these operations give rise to the transfer of the input focus. The passing of focus among windows can take place within one application or among different applications. Whatever the case, PM allows just one window to be active at any given

moment. The activation is indicated by a colored titlebar (a green one for standard configuration PM applications), and by controlling and handling all keyboard input operation. All these operations are performed automatically by PM according to a pre-set logic. The transfer of focus affects both the frame window and the client window. The message flow that is issued is different for the two windows. In general, you will be interested in knowing what happens at the window procedure level of the class to which the client window belongs, although it is often necessary to resort to the frame's subclassing to implement WPS compliant applications. A client window that acquires the input focus will receive all the following messages:

```
WM_FOCUSCHANGE
WM_REALIZEPALETTE
WM_ACTIVATE
WM_SETSELECTION
WM_SETFOCUS
```

and, when it loses it, it will be interested in the arrival of the following messages:

```
WM_SETFOCUS
WM_SETSELECTION
WM_ACTIVATE
```

It is quite natural to wonder whether the messages are first issued by PM to the window that loses focus or to that one that receives it. Common sense—and a good deal of system intelligence and espionage—indicates that messages will first reach the window to be deactivated, and only afterward will they get to the window to be activated. This solution prevents two windows being active at the same time, which would be a visually unpleasant and confusing solution.

It is quite interesting to examine in detail the contents of the messages involved in the focus transfer process, because this knowledge can be very useful in writing one's own code. The WM\_SETFOCUS message contains in mp1 the handle of the window that is about to acquire/lose the input focus. The message's status is denoted in the first short value present in mp2. When WM\_SETFOCUS is received by a window that is about to become active, then fFocus is equal to TRUE and the handle indicated in mp1 identifies the window that is losing the focus:

WM_SETFOCUS	0x000f	<i>Description</i>
mp1	HWND hwnd	Window handle losing the focus
mp2	SHORT fFocus	TRUE
Return Value	Reserved	

Instead, in the case that a WM\_SETFOCUS message is received by a window that is about to lose the input focus, then the first short in mp2 takes on the value of FALSE, and in mp1 there will be the handle of the window that will receive the focus:

WM_SETFOCUS	0x000f	<i>Description</i>
mp1	HWND hwnd	Window handle receiving the focus
mp2	SHORT fFocus	FALSE
Return Value	Reserved	

The return value of this message is a ULONG that theoretically is reserved, according to what is stated in the documentation. Actually, the handle of the window that is about to acquire or lose the focus is always returned.

The WM\_SETSELECTION message, contains in the first SHORT of mp1 a boolean value indicating the status of this selection: TRUE or FALSE. In practice, this message will indicate the status of the window with respect to the selection process. Both the return value and mp2 contain reserved values:

WM_SETSELECTION	0x0010	<i>Description</i>
mp1	USHORT usSelection	Selection flag (TRUE/FALSE)
mp2	Reserved	
Return Value	Reserved	

```
WM_SETSELECTION
fSelect = (BOOL) SHORT1FROMMP( mp1 ) ;
```

WM\_ACTIVATE always contains in mp2 the handle of the frame window of the window that is about to be activated/deactivated; in practice, it's its parent window. In the first short of mp1 you will have the activation status: TRUE or FALSE. The return value is always reserved, but, as we saw, it will always contain the handle of the window that is being acted upon.

WM_ACTIVATE	0x000d	<i>Description</i>
mp1	USHORT usActive	Activation flag (TRUE/FALSE)
mp2	HWND hwnd	Frame window handle
Return Value	Reserved	

The three messages WM\_SETFOCUS, WM\_SETSELECTION, and WM\_ACTIVATE correspond to these three actions:

- Transfer of the focus
- Selection of a window
- Activation

Often, a design objective is that of getting to know the handle of the window that has relinquished focus, or that is about to receive it. The function *WinQueryWindow()* allows you to explore the list of windows that might be affected by setting the QW\_NEXT, QW\_PREV, QW\_NEXTTOP, and QW\_PREVTOP flags. However, when focus is transferred to a newly created window, *WinQueryWindow()* doesn't work at all.

From this point of view, intercepting the WM\_ACTIVATE message is completely useless, because the only handle it can return is that of the application's frame window.

On the other hand, much more useful is the inspection of the contents of mp1 and mp2 in the WM\_SETFOCUS message. It is important to stress that it is not advisable to

modify the contents of `mp1` and `mp2` in this message, lest unstable system conditions arise. The designer is limited to inspecting the handle while it is passing by without changing its value.

There is yet another message involved in the transfer of focus, the `WM_FOCUSCHANGE` message. This message will not reach the application's window procedure, but is addressed only to windows like the frame window. To access this latter function there is no other means than that of resorting to its *subclassing*, a subject that will be dealt with in Chapter 10.

Knowing which window is being activated is of critical importance under many circumstances. When writing customized applications, you might want to prevent inexperienced users from cruising through the whole system, and maybe causing serious damage like erasing some files or just moving somewhere else in the tree-structured disk directory.

Quite simply, in some dialog windows you will want to make sure that the focus can be transferred (and this will be possible only by using the mouse) to some other window, only after some critical data has been provided. The knowledge of the window to which focus has been assigned before any system operation is undertaken is certainly useful. In Chapter 10 this subject will be explained in more systematic terms with some sample code.

For the time being, see how `IPMD` can be used on the `SETFOCUS` exercise to discover which values are contained in the `WM_SETFOCUS` message when a window is activated or deactivated. Listing 4.9 contains the source code of `OWNER.C` with changes in the window procedure that present a case statement for `WM_SETFOCUS`.

To verify the contents of `mp1` and `mp2` inside `WM_SETFOCUS` it is adequate to set a breakpoint on the first code statement after the `WM_SETFOCUS` case, and then run the program from within `IPMD` by pressing the right mouse button or one of the accelerators that run the application in Trace mode. Figure 4.18 shows what happens when focus is passed from the application window to another one.

This is a first, very important consideration: The activation or deactivation of a window takes place first at the frame window level, and not at the client window level. The entire process is the result of the interaction between the frame and all its controls. Often, in fact, the user will press the left mouse button on a window's titlebar to perform the selection.

This intelligence operation, extended to the `WM_ACTIVATE` message, manifests how the handle contained in `mp2` (identified by `0x5b2056F8`) is always the same as the handle of the parent window, the frame window of the application (identified by `hwndFrame`). This means that the `WM_ACTIVATE` message always contains in `mp2` the handle of the parent window of the client window, and never, as the documentation would have you believe, the handle of the window being activated or deactivated. In any case, the information in `mp2` is of little interest, because it is accessible to the application in several other ways. `WM_ACTIVATE` never informs the application about the handle of the window that is being activated, but simply indicates to the client window that its frame is about to be activated or deactivated. There is nothing wrong with this, it's just useless!



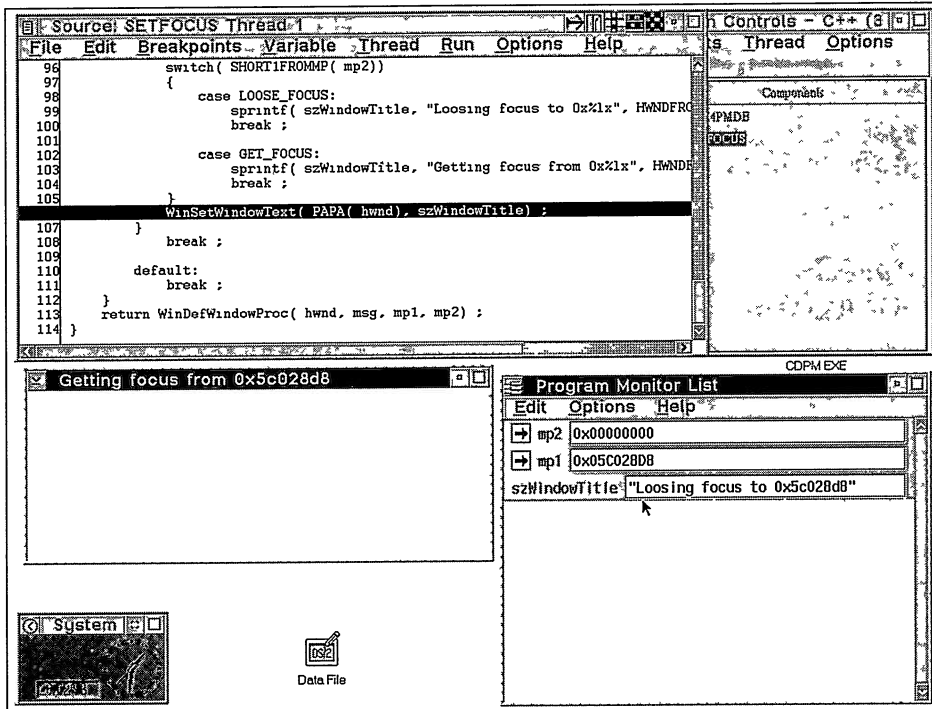


Figure 4.18 Values of identifiers in OWNER.C as displayed by IPMD.

Catching the `WM_ACTIVATE` message is useful from another point of view. Since the contents of `mp2` is the parent window's handle, it actually refers to the frame window. This information is not generally available in the window procedure of the class registered in the application, because the operation is performed in `main()`. If ever you were interested in knowing the handle of the frame window for some immediate or (more likely) subsequent purpose, there is nothing else you can do other than intercepting `WM_ACTIVATE` and storing the contents of `mp2` in a static class identifier.

## Adding a System Icon

In Chapter 5 we will see how to assign a user-defined icon to a generic window. Without dealing in this chapter with all problems related to resource files, you can display an icon on the screen whenever a window is minimized. You just have to reach out for any of the icons readily available in PM. To do this, send the frame window a `WM_SETICON` message that takes on the following syntax:

```
...
WinSendMessage( hwndFrame, WM_SETICON, MPFROMLONG( hptrIcon), 0L );
...
```

In mp1 you must always indicate the handle of a pointer or of an icon. In PM there are several bitmaps that you can borrow for this purpose; to obtain a handle, though, you will have to use the *WinQuerySysPointer()* function:

```
#define INCL_WINPOINTERS
HPOINTER APIENTRY WinQuerySysPointer( HWND hwndDesktop,
                                     SHORT iptr,
                                     BOOL fCopy) ;
```

<i>Parameter</i>	<i>Description</i>
hwndDesktop	Handle of the desktop or just HWND_DESKTOP
iptr	One of the defines listed in Table 4.13
fCopy	Indicates whether you want to copy the system image or access it directly

<i>Return Value</i>	<i>Description</i>
HPOINTER	Handle of the icon retrieved from the system's icons

The first parameter is always a handle to the desktop window (HWND\_DESKTOP), the second one is an index that allows you to select a pointer among those available in PM (Table 4.13).

**Table 4.13 Predefined PM Icons and Pointers That Can Be Retrieved with the *WinQuerySysPointer()* Function**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
SPTR_ARROW	1	Traditional left slanting arrow.
SPTR_TEXT	2	I-beam cursor; it is used by default in WC_ENTRYFIELD and WC_MLE class windows.
SPTR_WAIT	3	Hourglass displayed when an application performs a lengthy operation.
SPTR_SIZE	4	Cursor that is displayed when you select with the keyboard the Size option of the system menu.
SPTR_MOVE	5	Cursor that is displayed when you select with the keyboard the Move option from the system menu.
SPTR_SIZENWSE	6	Double-headed arrow pointing northwest and southeast; cursor that is displayed when you are resizing a window by positioning the cursor on the upper left or lower right-hand corner.

(continued)

**Table 4.13 (Continued)**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
SPTR_SIZENESW	7	Double-headed arrow pointing northeast and southwest; cursor that is displayed when you are resizing a window by positioning the cursor on the upper right or lower left-hand corner.
SPTR_SIZEWE	8	Double-headed arrow pointing horizontally; cursor that is displayed when resizing a window positioning the cursor on one of its two vertical borders.
SPTR_SIZENS	9	Double-headed arrow pointing vertically; cursor that is displayed when resizing a window positioning the cursor on one of its two horizontal borders.
SPTR_APPICON	10	Icon of a standard PM application.
SPTR_ICONINFORMATION	11	Icon with the shape of an information point.
SPTR_ICONQUESTION	12	Icon with the shape of a question mark.
SPTR_ICONERROR	13	Icon with the shape of an exclamation point.
SPTR_ICONWARNING	14	Icon with the shape of an exclamation point inside a triangle.
SPTR_CPTR	14	Icon with the shape of a stop sign.
SPTR_ILLEGAL	18	Icon with the shape of a forbidden sign; it is displayed in the File Manager application when you try to drag a file with the mouse to some other window that is not a directory.
SPTR_FILE	19	Icon with the shape of a file; used by the File Manager application to represent the contents of a directory.
SPTR_FOLDER	20	Icon with the shape of a folder.
SPTR_MULTIFILE	21	Icon representing multiple files; it is displayed by the File Manager when performing multiple file copies with the mouse.
SPTR_PROGRAM	22	Icon with the shape of a program.

In versions of OS/2 prior to 2.x other defines were used in place of those listed above:

SPTR_HANDICON	SPTR_ICONERROR
SPTR_QUESTICON	SPTR_ICONQUESTION
SPTR_BANGICON	SPTR_ICONWARNING
SPTR_NOTEICON	SPTR_ICONINFORMATION

The value returned by *WinQuerySysPointer()* is appropriately manipulated by the `MPFROMLONG` macro so that it can play the role of `mp1` in a `WM_SETICON` message:

```

...
WinSendMsg( hwndFrame, WM_SETICON,
            (MPARAM)WinQuerySysPointer( HWND_DESKTOP,
                                       SPTR_APPICON, FALSE),
            0L) ;
...

```

This portion of the code generally comes before the message loop; the minimization of the window will involve the drawing of a white icon with a light border along its perimeter.



# Input Tools and Resources

In PM there are three input sources: the keyboard, the mouse, and the timer. As opposed to what happens in character-based user interface systems, the keyboard—the traditional means of input—loses a great deal of its importance in the PM development model, to the benefit of the mouse. When you need to provide an application with your name or other information, the keyboard is matchless, maybe with the exception of pen-based input (OS/2 2.1 *pen extensions* will not be treated in this book). Often, keyboard input is performed by pressing special keys like Ins, Del, or the function keys. Although this kind of input can be performed with the keyboard, it can also be performed automatically by PM thanks to accelerators, menus (the main communication tool between the user and the application), and predefined classes like WC\_ENTRYFIELD and WC\_MLE.

The three kernel subsystems of OS/2—video, keyboard, and mouse—are completely disabled by PM, which resorts to its own way of handling and getting the user's input actions (Figure 5.1).

One of the principal problems inherent with the use of input tools is that of the uniqueness of the input source for the many applications that can be running simultaneously. This problem is handled by PM. The whole information flow produced by the user is converted into one or more specific messages which are passed to the window procedure of the class to which the active window belongs. According to PM's rules, only one window at a time can have the *input focus*, that is the control over all input tools. Actually, this consideration is true only for input originated by the keyboard. Pressing the A key will generate the WM\_CHAR message in the application's message queue, and provide a truly unique connection between the window and the keyboard (Figure 5.2).

The mouse, on the other hand, is exempt from these limitations. It acts almost independently, and can even send messages to windows that do not possess the input focus.

---

## The Keyboard

The keyboard plays a fundamental role in the interaction between the user and the application. Often, though, the designer may ignore the chore of handling the key-

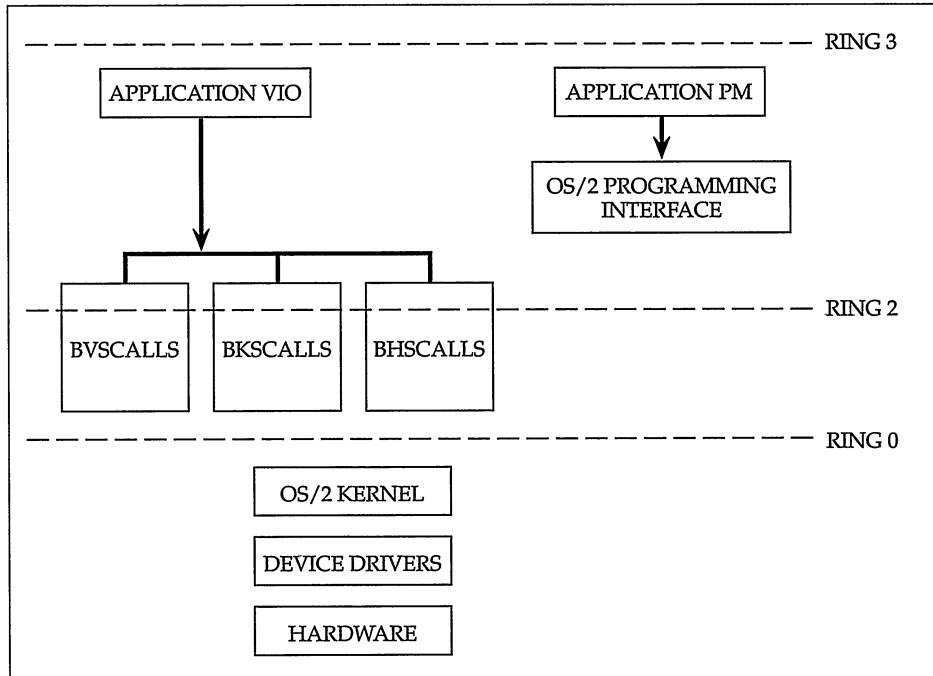
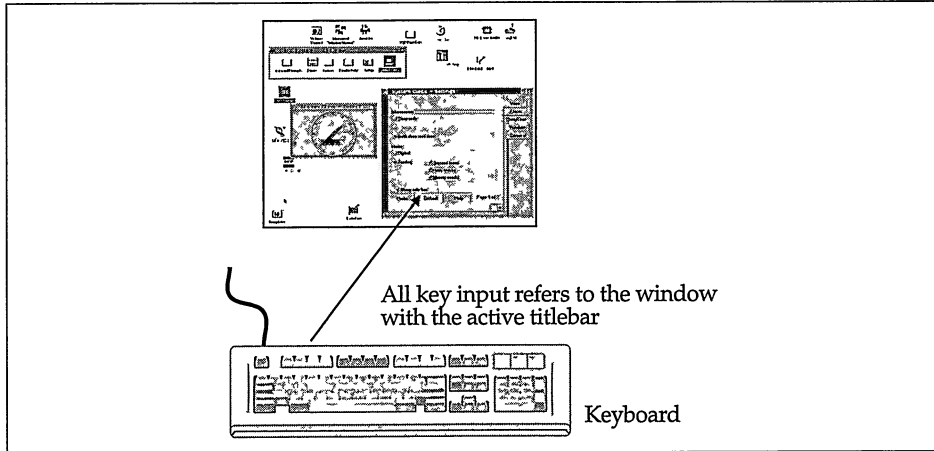


Figure 5.1 The structure of input subsystems in the OS/2 kernel and in OS/2 PM.

board. This is possible because PM can handle directly a great deal of actions performed with the keyboard, without the programmer having to write a single line of code. Furthermore, the presence of the `WC_ENTRYFIELD` and `WC_MLE` window classes will most often render useless processing input like a sequence of `WM_CHAR` messages, because everything is already being handled at the window procedure level of the appropriate classes.

Pressing a key will cause PM to insert a `WM_CHAR` message in the queue of the application that is active at that very moment (that has the input focus). Both PM's automatic processing, as well as what is coded in the window procedures of the `WC_MLE` and `WC_ENTRYFIELD` classes is fundamentally based on reading the contents of a `WM_CHAR` message that has the following structure:

<code>WM_CHAR</code>	0x007a	<i>Description</i>
<code>mp1</code>	<code>USHORT fs</code>	Keyboard control code
	<code>UCHAR cRepeat</code>	Repetition count
	<code>UCHAR scancode</code>	Hardware scan code
<code>mp2</code>	<code>USHORT chr</code>	Character code
	<code>USHORT vkek</code>	Virtual key code
Return Value	<code>BOOL fresult</code>	Flag for the message's processing



**Figure 5.2** Keyboard input is always addressed to the active window in the PM screen group.

The specific information provided WM\_CHAR completely fits in the 64 bits available in the pair mp1 and mp2. The flag fs identifies the control code of the keyboard. The value that this parameter can take is described in Table 5.1.

In general, the interpretation of what is keyed in at the keyboard happens by catching the WM\_CHAR message. PM's API provides you with the CHARMSG macro that lets you easily extract the values contained in WM\_CHAR.

**Table 5.1** Keyboard Control Codes as Defined by the fsFlags Identifier

<i>Control Code</i>	<i>Value</i>	<i>Description</i>
KC_CHAR	0x0001	The data contained in chr refers to one single character; otherwise the whole mp2 is null.
KC_VIRTUALKEY	0x0002	The data contained in vKey is valid; otherwise vKey is zero.
KC_SCANCODE	0x0004	The data contained in scancode is valid; otherwise the value of scancode is zero.
KC_SHIFT	0x0008	Flags the pressing of the SHIFT key.
KC_CTRL	0x0010	Flags the pressing of the CTRL key.
KC_ALT	0x0020	Flags the pressing of the ALT key.
KC_KEYUP	0x0040	The intercepted event referred to the releasing of a key. There is nothing similar for the pressing of a key, but you can check for the absence of the KC_KEYUP value.

*(continued)*

**Table 5.1 (Continued)**

<i>Control Code</i>	<i>Value</i>	<i>Description</i>
KC_PREVDOWN	0x0080	The key was pressed even before; otherwise it was not.
KC_LONEKEY	0x0100	Flags whether only one key was pressed and released and that in the meantime the user pressed no other key.
KC_DEADKEY	0x0200	The character code refers to a key that cannot be displayed on the screen.
KC_COMPOSITE	0x0400	The returned character is given by the combination of the current key with the previous dead key.
KC_INVALIDCOMP	0x0800	The current character is not valid in combination with the previous dead key.

```
#define CHARMSG(pmsg) ((PCHRMSG)((PBYTE)pmsg + sizeof(MPARAM)))
```

The macro extracts information from mp1 and mp2 on the basis of the CHARMSG structure:

```
typedef struct _CHARMSG
{ // charmsg
  USHORT fs ;          // mp1
  UCHAR cRepeat ;
  UCHAR scancode ;
  USHORT chr ;        // mp2
  USHORT vkey ;
} CHRMSG ;

typedef CHRMSG *PCHRMSG ;
```

The following code fragment shows typical use of the CHARMSG macro inside a window procedure to check for the KC\_CHAR flag that provides the ASCII code of the character being keyed in at the keyboard.

```
...
switch( msg)
{
  case WM_CHAR:
  {
    if( CHARMSG( &msg) - fs & KC_CHAR)
    {
      switch( CHARMSG( &msg) - chr)
      {
```

```

        ...
    }
    ...
}
break ;

...
}
...

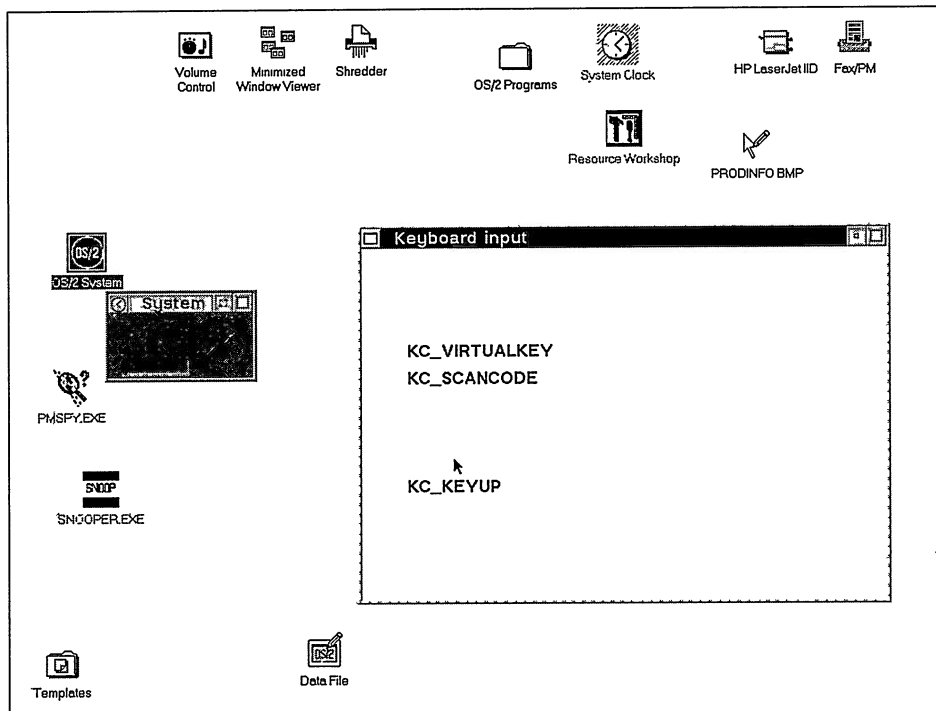
```

The application will check the `KC_KEYUP` code to decide if a key has been pressed or released, then acts according to the value of the key that was pressed.

## Keyboard Usage



Let's examine a simple sample program, `KBD` (Listing 5.1 and Figure 5.3), displaying in its client window several pieces of information relating to the key that was pressed. All output activity is concentrated in the `WM_PAINT` message's processing, so that



**Figure 5.3** The application intercepts all characters coming from the keyboard and displays them in its own client window.

output will be performed optimally even if the window gets resized or covered by other active application windows in the screen group.

We will see better ways of using the WM\_CHAR message in Chapter 10, after we learn more about *subclassing*.

---

## The Mouse

Any mouse movement on the desktop has a corresponding WM\_MOUSEMOVE message. Different from WM\_CHAR, the WM\_MOUSEMOVE message will always and only reach the window procedure of the window where the *hot spot* was at that very moment. This greater freedom of the mouse with respect to the window having the input focus, explains why it can service several windows according to its position on the screen. Table 5.2 presents the complete list of messages generated by using the mouse.

Each of the messages generated by the mouse will contain in `mp2` the cursor's position, which will always be expressed in coordinates relative to the underlying window. As you can see from looking at Table 5.2, PM is capable of handling pointing devices that have as many as three different buttons. Button number one always

**Table 5.2 List of Messages Generated by the Mouse**

<i>Message</i>	<i>Value</i>	<i>Description</i>
WM_MOUSEMOVE	0x0070	Generated any time the mouse is moved.
WM_BUTTON1DOWN	0x0071	Generated when the left mouse button is pressed.
WM_BUTTON1UP	0x0072	Generated when the left mouse button is released.
WM_BUTTON1DBLCLK	0x0073	Generated when the left mouse button is double-clicked: The time interval for detecting a double-click can be set through the control panel.
WM_BUTTON2DOWN	0x0074	Generated when the right mouse button is pressed.
WM_BUTTON2UP	0x0075	Generated when the right mouse button is released.
WM_BUTTON2DBLCLK	0x0076	Generated when the right mouse button is double-clicked: The time interval for detecting a double-click can be set through the control panel.
WM_BUTTON3DOWN	0x0077	Generated when the center mouse button is pressed.
WM_BUTTON3UP	0x0078	Generated when the center mouse button is released.
WM_BUTTON3DBLCLK	0x0079	Generated when the center mouse button is double-clicked: The time interval for detecting a double-click can be set through the control panel.

---

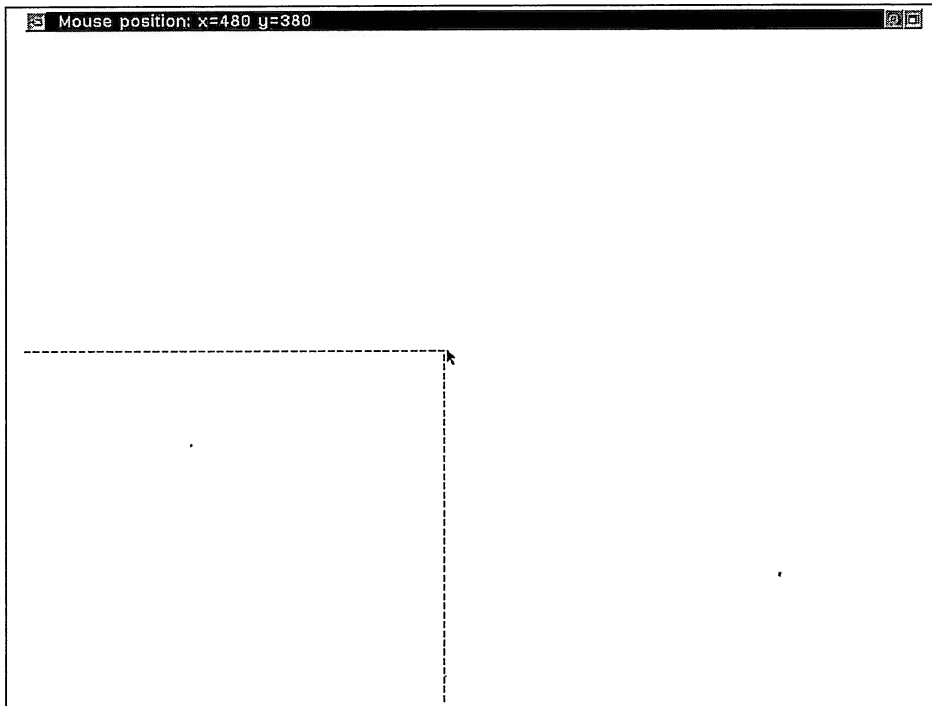
refers to the leftmost button; button number two to the rightmost button; and button number three to the center button, if one is present.

Even the slightest mouse movement will cause the system to send a `WM_MOUSEMOVE` message to the window under the mouse cursor. To test this, you can create a simple program that intercepts in its window procedure the `WM_MOUSEMOVE` message, retrieves the mouse's position from `mp2`, and displays the coordinate's values somewhere on the screen. To simplify the whole thing, let's just use the application's titlebar (described in Chapter 2) as the output area.

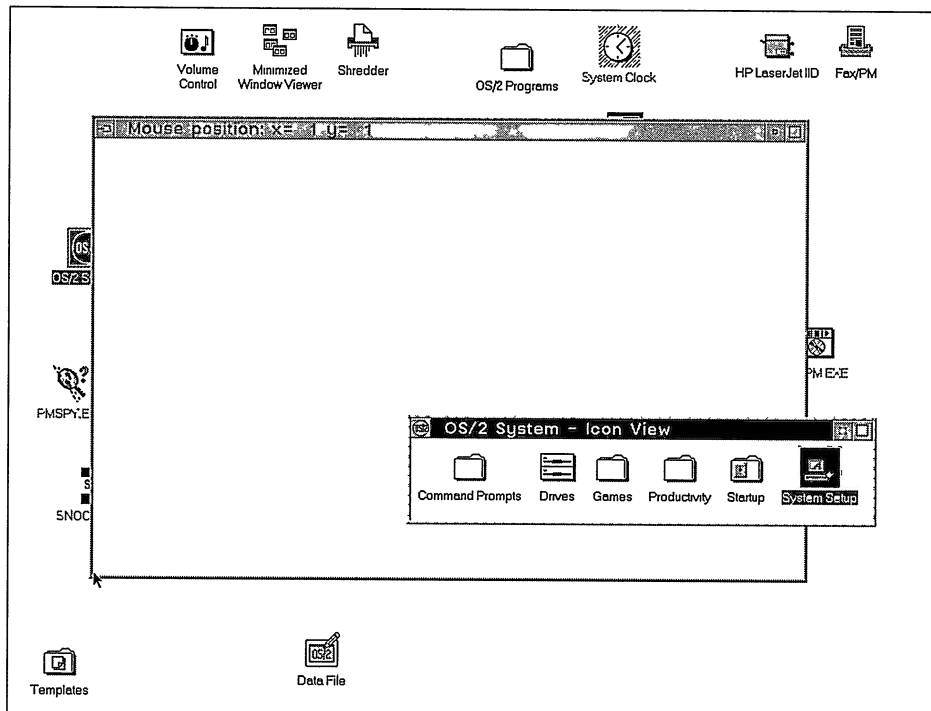
Figure 5.4 shows the mouse pointer inside the application's client window, while Figure 5.5 shows the independent nature of the input generated by the mouse.

Listing 5.2 refers to the application shown in Figures 5.4 and 5.5. In Listing 5.2 you will also see the messages `WM_BUTTON1DBLCLK` and `WM_BUTTON2DBLCLK`. In the code dealing with the double-click of the mouse's left button the function `WinSetCapture()` is called. This function "captures" the mouse. What this means is that the relationship between the mouse and the window indicated by the function's second parameter becomes exclusive.

```
#define INCL_WININPUT
BOOL WINAPI WinSetCapture( HWND hwndDesktop, HWND hwnd ) ;
```



**Figure 5.4** The mouse's position is always expressed in coordinates relative to the bottom left corner of the client window.



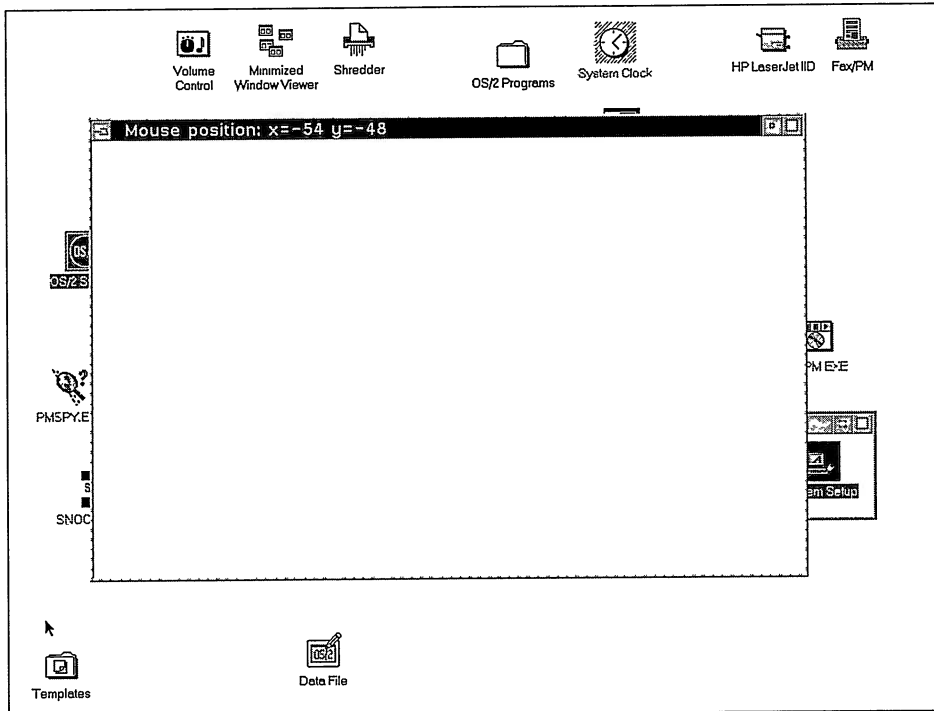
**Figure 5.5** The application that displays the mouse's coordinates is disabled (the focus is on OS/2 System), although the WM\_MOUSEMOVE always gets to its destination.

<i>Parameter</i>	<i>Description</i>
hwndDesktop	The desktop handle, HWND_DESKTOP
hwnd	Handle of the window that needs to capture the mouse input or NULLHANDLE if you need to release the mouse capture
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The first handle will always indicate the desktop, while the second one refers to the window that will become the exclusive owner of any kind of mouse event—movements and pressing of mouse buttons. The second handle can be replaced by the special value `HWND_THREADCAPTURE` to signal that the capture of the mouse does not refer to a single window, but to all windows present in the thread that has called the function. In Figure 5.6 you can see how the titlebar now displays even negative values, which indicate negative coordinates with respect to the origin in the client window, which is always the lower left-hand side corner.

The coordinate system of a PM window will always have the origin in the lower left-hand corner. Increasing values on the X axis go from left to right, and from down





**Figure 5.6** The titlebar displays negative values when the mouse is outside of the client window and the mouse has been “captured.”

to up on the Y axis. The presence of the mouse to the left and below the lower left-hand side corner of the window produced by Listing 5.2 is the condition necessary in order to see negative coordinate values.

There is no specific function for breaking the exclusive relationship between a captured mouse and a window. Call the same *WinSetCapture()* function, but specify *NULL-HANDLE* as its second parameter. In Listing 5.2 this happens in the code handling double-clicks of the mouse’s right button. To know which window has captured the mouse, you need to call the *WinQueryCapture()* function, which has the following syntax:

```
#define INCL_WININPUT
HWND WINAPI WinQueryCapture( HWND hwndDesktop );
```

**Parameter**            **Description**

hwndDesktop            Handle of the desktop, *HWND\_DESKTOP*

**Return Value**        **Description**

HWND                    Handle of the window that has captured the mouse or *NULL-HANDLE* if the mouse is free

Listing 5.2 also includes the interception of the *WM\_BUTTON1DOWN* and *WM\_BUTTON2DOWN* message; that is, when the left and right buttons are pressed. When one of

these events take place, the application will see if the hot spot is over some other window. To determine this, some intermediate operations are needed to convert the coordinates and to get the window's handle. The position of the mouse can always be converted between the coordinates of two generic windows with the *WinMapWindowPoints()* function:

```
#define INCL_WINWINDOWMGR
BOOL APIENTRY WinMapWindowPoints(  HWND hwndFrom,
                                   HWND hwndTo,
                                   PPOINTL prgptl,
                                   LONG cwpt) ;
```

<i>Parameter</i>	<i>Description</i>
hwndFrom	Handle of the window to which the points in the pptl parameter refer
hwndTo	Handle of the target window for the conversion
prgptl	Array of POINTL structures containing points to be converted
cwpt	Number of points contained in the prgptl array
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The first handle identifies the window referred to by the points in the POINTL array, that is, the third parameter. The second handle identifies the reference window. By specifying NULLHANDLE or HWND\_DESKTOP in the first case, the points used by the application will be in screen coordinates. If the second parameter is set to NULLHANDLE or HWND\_DESKTOP, the conversion is from the window to the desktop. A third possibility is between two windows in the same application or in different programs. The third parameter is an array of points expressed like a series of POINTL or RECTL structures. The last parameter indicates the number of points in the array (if you are using an array of RECTL structures the number of points has to be doubled).

In Listing 5.2 the *WinMapWindowPoints()* function is used to convert the mouse position into screen coordinates. The result will always be positive values, whether or not the initial values were positive. Once you know the mouse's position in the screen, you need to answer some more questions. First, you have to check which is the window underlying the coordinates (x, y) returned by *WinMapWindowPoints()*. If it is the application's own window or if it is the desktop window, then processing will pass on the WM\_BUTTONXDOWN message to the default window procedure for its final processing. If the mouse is over pixels taken by some other window, then the application will send an activation message to that window, so that the input focus will be transferred to it. To check for the presence of a window, you will need to call *WinWindowFromPoint()*.

```
#define INCL_WINWINDOWMGR
HWND APIENTRY WinWindowFromPoint(  HWND hwnd,
                                    PPOINTL pptl,
                                    BOOL fChildren) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window to inspect
pptl	Coordinates of the point, expressed relative to the window indicated by the first parameter
fChildren	Checks for first generation child windows (FALSE) or subsequent generations (TRUE)
<i>Return Value</i>	<i>Description</i>
HWND	Handle of the window underlying the mouse's <i>hot-spot</i> or NULLHANDLE in case of error

The returned value is the handle of the window that occupies the position indicated by the point given to the function, or NULLHANDLE in case of errors. The whole screen of OS/2 2.1 systems is occupied by WPS, a window belonging to the class WC\_CONTAINER. Therefore, if the parameters to *WinWindowFromPoint()* are correct, you will always obtain a window handle.

The handling of the WM\_BUTTONxDOWN message is abandoned if the returned handle is null (error in the call), or if it corresponds to the application's client or frame window or to the desktop window. In any other case, you will have to release the mouse capture and convert the mouse's hot-spot position into coordinates relative to the window that has been detected. This operation can be performed with *WinMapWindowPoints()*, which will let you pass the WM\_BUTTONxDOWN message to the window that has been selected by the mouse, by specifying the correct values for the mouse's position. Thus, you are actually simulating the operations of selecting and activating a window when the mouse is not captured, and leaving to the system the task of executing the various phases of releasing and acquiring the input focus.

The procedure just described is the working principle behind applications like PMSPY.EXE that allow the user to select directly the window to be inspected with the pointing device. By examining the mouse messages listed in Table 5.2, it is easy to conclude that the only actions that are interceptable are the pressing and releasing of the mouse buttons and the moving of the pointing device on the screen. The interaction between the mouse and the titlebar, the sizing border, and the menus is handled automatically by PM; this is always done, however, on the basis of the messages listed in Table 5.2, with no work whatsoever for the programmer. The best way to get the data describing the interaction of the mouse with these windows is to exploit their *subclassing* (see Chapter 10).

## *Teaching an Old Mouse New Mouse Tricks*

In OS/2 2.1 the mouse is capable of generating many more messages than in the past. These messages are treated separately from those in Table 5.3, to stress the differences between the standard behavior and that induced by WPS.

In OS/2 2.1 the mouse becomes the principal tool for interacting with the system. In addition, the traditional actions, like selection (single-click) and selection/confirmation

**Table 5.3 Mouse Messages Regarding Selections and Drag & Drop Operations**

<i>Message</i>	<i>Value</i>	<i>Description</i>
WM_BUTTON1MOTIONSTART	0x0411	Issued after pressing the right mouse button and moving the mouse from its starting position.
WM_BUTTON1MOTIONEND	0x0412	Issued after releasing the right mouse button for completing a previous WM_BUTTON1-MOTIONSTART.
WM_BUTTON1CLICK	0x0413	Issued after a press/release action on the right mouse button with no intervening movement.
WM_BUTTON2MOTIONSTART	0x0414	Issued after pressing the left mouse button and moving the mouse from its starting position.
WM_BUTTON2MOTIONEND	0x0415	Issued after releasing the left mouse button for completing a previous WM_BUTTON2-MOTIONSTART.
WM_BUTTON2CLICK	0x0416	Issued after a press/release action on the left mouse button with no intervening movement.
WM_BUTTON3MOTIONSTART	0x0417	Issued after pressing the center mouse button and moving the mouse from its starting position.
WM_BUTTON3MOTIONEND	0x0418	Issued after releasing the center mouse button for completing a previous WM_BUTTON3MOTIONSTART.
WM_BUTTON3CLICK	0x0419	Issued after a press/release action on the center mouse button with no intervening movement.
WM_BEGINDRAG	0x0420	Issued after a WM_BUTTONxMOTIONSTART.
WM_ENDDRAG	0x0421	Issued after a WM_BUTTONxMOTIONEND.
WM_SINGLESELECT	0x0422	Posted when the user selects an object.
WM_OPEN	0x0423	Posted when an object is opened in WPS.
WM_CONTEXTMENU	0x0424	Issued after a WM_BUTTONxCLICK.
WM_CONTEXTHELP	0x0425	
WM_TEXTEDIT	0x0426	Posted when an object is renamed by pressing the left button and the ALT key.
WM_BEGINSELECT	0x0427	Issued after a WM_BUTTON1MOTIONSTART when the user keeps the right mouse button depressed for performing a selection.
WM_ENDSELECT	0x0428	Issued after a WM_BUTTON1MOTIONEND to complete a selection operation.

(double-click), the mouse also allows you to click, select some object in a container, and then perform a drag & drop operation.

## Mouse Clicking

The normal interaction with a mouse button includes receiving a message when a button is pressed and a message when it is released. Often, the programmer will catch only one of these two events for implementing his or her own logic in the code. The presence of the WM\_BUTTONNCLICK message allows for a more complex treatment. In fact, now it is very easy to distinguish between a simple button pressure (*clicking*) and the pressing or releasing of a button. When the designer needs to code a mouse interaction, he can choose to use WM\_BUTTONNCLICK, and then use WM\_BUTTONNDOWN and WM\_BUTTONNUP in special cases.

		<i>Description</i>
WM_BUTTONNCLICK	0x0413	
mp1	POINTS ptspointerpos	Mouse's position in the window
mp2	USHORT fshittestres	Mouse's position
	USHORT fsflags	Keyboard control code
Return Value	BOOL fResult	Message was processed (TRUE) or not (FALSE)

The POINTS structure is made up of a pair of short values containing the mouse's coordinates expressed with respect to the window's lower left-hand corner. The WM\_CONTEXTMENU message completes the clicking sequence.

		<i>Description</i>
WM_CONTEXTMENU	0x0424	
mp1	POINTS ptspointerpos	Mouse's position in the window
mp2	USHORT usPointer	Indicates input from the mouse (TRUE) or from the keyboard (FALSE)
Return Value	BOOL fResult	Message was processed (TRUE) or not (FALSE)

This message is always posted, and is used to display the *window context menu*, that is, the menu associated with an object, a typical feature of WPS.

## Selection of an Object

There are several ways of selecting one or more objects in a container or in WPS. The simplest and most intuitive is that of positioning the mouse pointer on an icon and pressing the left mouse button. This action will cause the appearance in the appropriate window procedure of the following sequence of messages:

```
WM_BUTTONNDOWN
...
WM_BUTTONNUP
WM_BUTTONNCLICK
WM_SINGLESELECT
```

The physical action performed on the mouse will first generate a WM\_BUTTON1CLICK, immediately followed by a WM\_SINGLESELECT.

		<i>Description</i>
WM_SINGLESELECT	0x0422	
mp1	POINTS ptspointerpos	Mouse's position in the window
mp2	USHORT usPointer	Indicates input from the mouse (TRUE) or from the keyboard (FALSE)
Return Value	BOOL fResult	Message was processed (TRUE) or not (FALSE)

For instance, the application will receive the WM\_SINGLESELECT message for each object selected in a container.

## Selecting More Objects

The message flow received by an application because of the simultaneous selection through the mouse of several objects is not much more complex than what we have just seen. The fundamental difference with respect to the previous two cases is that you need to perform two actions with the mouse:

- Press the left button
- Move the mouse pointer

The first action will generate the WM\_BUTTON1DOWN message. In this case, however, the user will keep the mouse button down while moving the mouse on the screen. Immediately after the first movement of the mouse, the whole logic of clicking is rendered moot, because it is based on the hot spot being stationary over the same position both for the pressing and the releasing of the mouse button. Therefore, the code will receive the WM\_BUTTON1MOTIONSTART message, which signals the start of mouse movement.

		<i>Description</i>
WM_BUTTON1MOTIONSTART	0x0411	
mp1	POINTS ptspointerpos	Indicates mouse's position in the window
mp2	USHORT usPointer	Indicates input from the mouse (TRUE) or from the keyboard (FALSE)
Return Value	BOOL fResult	Message was processed (TRUE) or not (FALSE)

This message is then immediately followed by WM\_BEGINSELECT. On the screen you will see what is shown in Figure 5.7.

When the right mouse button is released, the messages WM\_BUTTON1UP, WM\_BUTTON1MOTIONEND, and WM\_ENDSELECTION will show up in the application's queue. To summarize, the sequence is as follows:

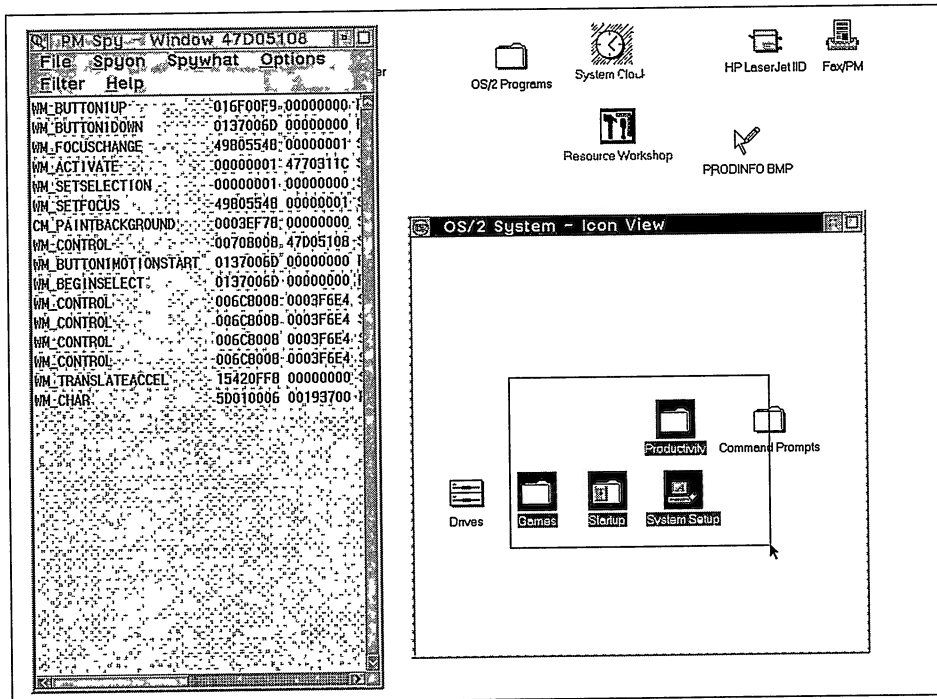


Figure 5.7 The start of a selection operation with the mouse is indicated by the display of an extensible selection rectangle (marquee).

```

WM_BUTTON1DOWN
WM_BUTTON1MOTIONSTART
WM_BEGINSELECTION
...
WM_BUTTON1UP
WM_BUTTON1MOTIONEND
WM_ENDSELECTION

```

The pair of messages `WM_BUTTON1MOTIONSTART` and `WM_BUTTON1MOTIONEND` serve the fundamental purpose of allowing the programmer to perform on-screen changes to the object or the window with which the mouse is interacting. The actual selection logic is contained in the code dealing with intercepting the messages `WM_BEGINSELECTION` and `WM_ENDSELECTION`.

## Drag & Drop

Chapter 12 covers this subject, so you will have fun—if you get that far! What is interesting here with *drag & drop* is the flow of mouse messages that characterizes it. Dragging is an operation performed by default with the right mouse button (number 2), which is different from the default selection operation, performed with the left

button (number 1). This is the only basic difference that there is with respect to selection; in fact, the rest is just the same kind of homework, as you can see in the following list of messages:

```
WM_BUTTON2DOWN
WM_BUTTON2MOTIONSTART
WM_BEGINDRAG
...
WM_BUTTON2UP
WM_BUTTON2MOTIONEND
WM_ENDDRAG
```

The pair of messages WM\_BEGINDRAG and WM\_ENDDRAG delimit, from a logical point of view, a dragging operation on the screen. The syntax of WM\_BEGINSELECTION, WM\_ENDSELECTION, WM\_BEGINDRAG, and WM\_ENDDRAG is the same as that of WM\_SINGLESELECT.

## Timers

PM provides the software designer with up to 40 simultaneous timers. A timer is a kind of alarm clock that is capable of sending a signal at a preset time interval. It can be considered an input tool, because, if correctly used, it allows you to execute code fragments that are repeated with a preset frequency.

In OS/2 2.1 multitasking is *preemptive*, in PM it is not. The previous statement is true. If a PM application takes hold of the CPU for a long enough time, the user will no longer be able to interact with other objects in the WPS. This does not mean that the whole system is locked out. The other threads in PM will continue to operate correctly and be serviced by the processor according to the traditional preemption mechanism. Only the user/PM interaction is adversely affected. This problem is generally solved by writing multithreaded applications, but will probably find a true solution in forthcoming versions of the system, which will implement a new kind of input logic.

Dealing with the WM\_TIMER message may make you want to bypass this unpleasant feature of PM. However, the WM\_TIMER message has its own peculiar charms, and it turns out not to be smart enough to try to circumvent it.

To activate a timer, you have to indicate a trigger frequency. However, there is no guarantee that you will receive a message in the preset amount of time, for the very reason that there can be some locking condition in PM. When an application happens to be in such a situation, as soon as it gets the processor's attention, it will receive just one WM\_TIMER message, the one that is triggered at that moment. All preceding timer events that could not be perceived are lost. The application can compute how much time has passed since it received an earlier WM\_TIMER message by checking the system clock. The time difference between two successive WM\_TIMER messages divided by the frequency interval will give the number of WM\_TIMER messages that have not been detected. This is the logical framework that governs programs that deal with time.



WM_TIMER	0x0024	<i>Description</i>
mp1	USHORT idTimer	Timer's ID
mp2	Reserved	
Return Value	Reserved	

The only useful information that can be extracted from WM\_TIMER message is the ID of the originating timer, which can help to identify the source of this alarm. Many commercial editors use a timer to display the mouse cursor intermittently, even while the keyboard is being used. Both E.EXE and EPM.EXE implement such a solution.

To show the operation of PM's timers, let's write an application that will flash the frame controls of a window when you press the left mouse button. Pressing the right mouse button will stop the flashing. The function is *WinFlashWindow()*:

```
#define INCL_WINFRAMEMGR
BOOL APIENTRY WinFlashWindow( HWND hwndFrame, BOOL fFlash ) ;
```

<i>Parameter</i>	<i>Description</i>
hwndFrame	Handle of the frame window
fFlash	Activation/deactivation of the flashing
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

If the second parameter is set to TRUE, it will activate the flashing of the frame window. Flashing will go on until the function is called again with that parameter set to FALSE.

The use of a timer allows you to vary the frequency and duration of the window's flashing. When the left mouse button is pressed (a WM\_BUTTON1DOWN message is received) a timer is immediately activated with *WinStartTimer()*:

```
#define INCL_WINTIMER
ULONG APIENTRY WinStartTimer( HAB hab,
                              HWND hwnd,
                              ULONG idTimer,
                              ULONG dtTimeout ) ;
```

<i>Parameter</i>	<i>Description</i>
hab	Anchor block handle
hwnd	Handle of the window associated with the timer
idTimer	ID of the timer being started
dtTimeout	Timer frequency expressed in milliseconds
<i>Return Value</i>	<i>Description</i>
ULONG	Timer ID when the second parameter is set to NULLHANDLE

The anchor block handle is provided by the *WinQueryAnchorBlock()* function. The second parameter identifies the window that is the owner of the timer; it is the window procedure of this window that will receive the WM\_TIMER messages. The third parameter assigns an ID to the timer when it is created, so that later it will be

possible to identify the source of the WM\_TIMER message. The last parameter defines in milliseconds the time interval between one WM\_TIMER message and the next.

To stop a timer, you have to call *WinStopTimer()* specifying in the same order the first three parameters of *WinStartTimer()*:

```
#define INCL_WINTIMER
BOOL APIENTRY WinStopTimer( HAB hab,
                            HWND hwnd,
                            ULONG idTimer ) ;
```

<i>Parameter</i>	<i>Description</i>
hab	Anchor block handle
hwnd	Handle of the window associated with the timer
idTimer	ID of the timer being stopped
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

Once the time interval, indicated when the timer was created, has passed, the application will receive a WM\_TIMER message. It is good practice to test which timer generated a message by checking the timer's ID; this value is contained in mp1.

When the first WM\_TIMER message is received by the window procedure, you can activate the window's flashing by calling *WinFlashWindow()*. However, this is not the only operation that you need to perform. For the sample to work properly, you also need to activate a second timer that will obviously have a different ID and a time-out interval that is exactly half that of the first timer. In this way, the application will receive a second WM\_TIMER message, but this timer will have been issued by the second timer. By intercepting the WM\_TIMER message, you can then deactivate the window's flashing by once again calling *WinFlashWindow()*, but this time with the boolean parameter set to FALSE, and deactivating the second timer with *WinStopTimer()*.

The next WM\_TIMER received by the program will naturally have been issued by the only timer that is still active in the system—the first one—and therefore the whole process is repeated by activating the window's flashing. In this way you can vary the window's flashing frequency, exempting it from the default behavior provided by *WinFlashWindow()*. Listing 5.3 shows the entire source code of the sample timer program.




---

## The Resource File

The flow chart shown in Figure 2.4 (Chapter 2) emphasized one of the special characteristics of the development model for PM: the resource file. It is an ASCII text file that can be edited with a common text editor or word processor. It contains reference to text and/or binary resources. The syntax of an .RC file is quite similar to a C source file. Comments can be inserted with the classic character combinations /\* and \*/ or even with a double slash //. Most of the preprocessor directives are valid even for the RC.EXE resource compiler.

The description of resources is made through special directives that build up the syntax of this file. All information is then compiled with a resource compiler, a special tool that is provided with the development Toolkit that produces a file with the .RES extension. After being compiled, the resources need to be inserted in an appropriate section of the executable module previously generated by the linker. Resources are thus an integral part of an .EXE file, even if the process for creating them follows a completely different path from that required for the source code. During execution, an application will load the resources from this area in the EXE file. It can do so in two ways:

- Explicitly (the API provides several Win and Gpi functions like *WinLoadxxx* or *GpiLoadxxx*)
- Indirectly, with some particular API functions

In most applications, the resources contained in the executable module will satisfy the program's requirements. PM's API, however, also allows access to resources that reside in some other executable module, and even the creation of a DLL containing resources exclusively.

## The Nature of Resources

All resources are handled by the resource compiler to produce an object that is characterized by the information shown in Table 5.4.

**Table 5.4 Information Characterizing a Resource After It Has Been Loaded**

<i>Information</i>	<i>Description</i>
fResType	Indicates if the type of the resource identifier is an integer or a text string: In PM, only integer values are allowed. Therefore, the value of this piece of data is always 0xFF.
usResType	Specifies the type of the resource identifier with values in the range from 0 to 64Kb.
fResID	Indicates if the resource identifier is an integer or a text string: In PM only integer values are allowed. Therefore the value of this piece of data is always 0xFF.
resid	Indicates the resource ID. It is always a numeric value in the range from 0 to 64Kb.
fsOptions	Load options (PRELOAD or LOADONCALL) and operation for the resource's memory management (MOVEABLE, FIXED, or DISCARDABLE). PRELOAD takes the value of 0x0040, provided the resource is of the type LOADONCALL. MOVEABLE takes the value 0x0010, provided the resource is not of the type FIXED. DISCARDABLE is 0x1000.
cb	Size of the resource in bytes.
bytes	List of bytes of the resource.

The knowledge of the internal organization of the information of every resource is useful when you need to define specific resources for an application and optimize its loading during execution. In general, the designer can take advantage of the appropriate API functions that are available.

Let's now examine in detail the various resources supported by PM. Table 5.4 also lets us identify the various kinds of resources supported by OS/2 2.1.

## *The Text Resources*

This term indicates all those resources that can be described by typing in manually a template or a sequence of commands that are comprehensible to the RC.EXE compiler. These are generally very simple statements, most of them are lexical in nature and can be described by phrases in English. Table 5.5 lists all these resources in alphabetical order.

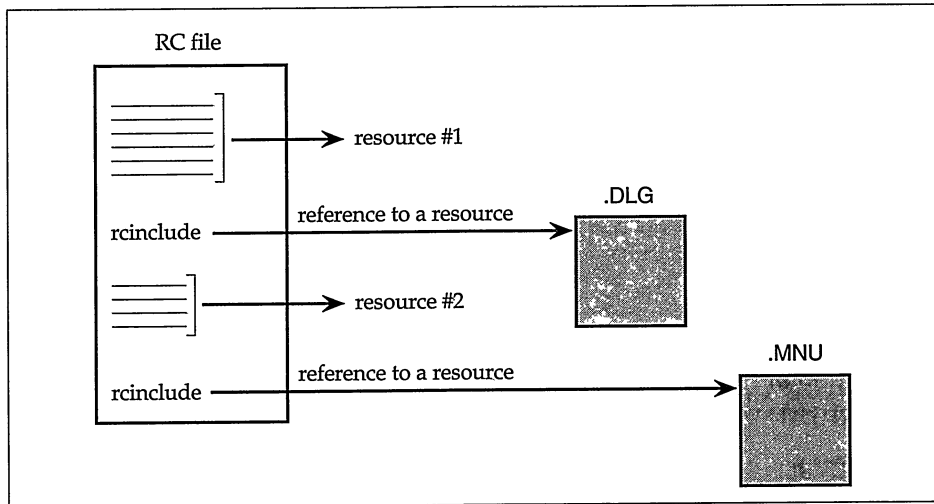
Very often a textual resource is described manually by the software designer, by typing its text directly in a file with the RC extension. At other times, it is preferable to generate standalone text files for each resource or group of correlated resources, and then collect all information directly inside a resource file (Figure 5.8). This case is typical of the dialog templates used for dialog windows.

The inclusion of a text file in a resource file is governed by the `rcinclude` directive, immediately followed by the name of the ASCII file to be included:

```
// TWENY.RC
#include <os2.h>
...
rcinclude TWENY.MNU
...
```

**Table 5.5 List of Textual Resources in a PM Resource File**

<i>Value</i>	<i>Description</i>
ACCELTABLE	Accelerator table.
ASSOCTABLE	Tables of associations of a category of files with another program.
DIALOG	Dialog template.
HELPTABLE	Help table.
MENU	Menu template.
MESSAGETABLE	Message strings.
PRESPARAMS	Definition of the presentation parameters.
STRINGTABLE	Text strings.
WINDOWTEMPLATE	Window template.



**Figure 5.8** The resource file and the other text files that can separately contain the description of a textual resource.

where TWENY.MNU contains, for instance, one or more menu templates. The preprocessor `#include` directive can also be used in place of `rcinclude`. The final result is the same, even though a less readable listing will be generated, because the `#include` is most often used only with files that have the extension `.H`.

```
// TWENY.RC
...
#include <os2.h>
...
#include "TWENY.MNU"
...
```

OS2.H is not strictly necessary, especially when the RC file is small or does not contain resources of the `ACCELTABLE` or `PRESPARAMS` types. In fact, it is only in this case that it is necessary to include OS2.H in order to access the defines of the virtual keys for defining the accelerator resources or the `PP_` definitions. An RC file easily supports multiple `rcinclude` directives, placed anywhere in the text, for the inclusion of different kinds of text objects: *dialog templates*, *menu templates*, and others. For the dialog templates, there exists also the specific `dlginclude` directive, which is used quite seldom because the more generic `rcinclude` or even `#include` is preferable.

Thus there exists only one resource file for application, even though this resource file can be made up of several other files with extensions such as `ICO`, `BMP`, `FNT`, `MNU`, `ACC`, and others.

Of all the various kinds of textual resources listed in Table 5.5, only `STRINGTABLE` and `MESSAGETABLE` are not associated with an ID identifier. This means that there can

be only one STRINGTABLE (or MESSAGE TABLE) area for each resource file. All other directives listed in Table 5.5 introduces objects that have IDs, and therefore can appear more than once in a resource file (PRESPARAMS are treated differently). Therefore, you can have multiple menu templates or multiple dialog templates, all coexisting without problems in an .RC file. In the next chapters we will analyze the menu templates and the dialog templates; for now, let's concentrate on the STRINGTABLE resource.

## STRINGTABLE

The STRINGTABLE directive (which can be typed in uppercase, lowercase, or a combination of the two) defines an area that will contain text strings. The syntax of this directive is very simple, as it only needs to delimit a block within a pair of braces.

```
STRINGTABLE
{
    StringID, "STRING"
}
```

Within the block you will need to list all text strings used by the application: error messages, information messages, names of classes you want to register in the program, titles to give to the windows, and so on. To sum up, in a STRINGTABLE you list all the strings displayed by the program. The syntax of STRINGTABLE requires that you type each text string within a pair of double quotes, and to precede each text string with a numeric ID.

```
// TWENY.RC
...
STRINGTABLE
{
    100, "Error Message"
}
```

The number (100 in the above example) is chosen by the programmer, provided it is in the range from 0 to 64K. The only thing to be careful about is that you must assign a unique ID, otherwise you will get an error message during compilation. The order in which you list the strings in the STRINGTABLE area does not affect the resource compiler's actions. A good criteria for defining strings is to group them in homogenous groups, and assign progressive IDs in each block. So, for instance, you could group all strings regarding error messages, all those for class names, those for window titles, and those for information messages. The first group of strings could have IDs in the range 1000–1100; the second group in the range 300–400; the third group between 500 and 550, and so on.

The IDs can be specified both in decimal as well as in hexadecimal notation. It is a common practice to define all IDs in the applications header file, rather than making

direct assignments as in the previous example. In the header file TWENY.H you might thus find the define ST\_ERR1 with the numerical value of 100:

```
// TWENY.H

...
#define ST_ERR1 100
...
```

A convention followed in this book is that of using the prefix ST\_ for categorizing all IDs used in the STRINGTABLE area, although you might adopt any convention you're comfortable with.

You can assign the same ID to different objects declared in a resource file, provided they are not of the same resource type (the resource compiler will signal two strings with the same ID as an error, but it will be happy if the same ID is shared between a STRINGTABLE string and a MENU menuitem).

Returning to the syntax of STRINGTABLE, the comma between the ID and the text string's opening double quote is optional.

By taking advantage of the header file, the STRINGTABLE area can be changed in the following way:

```
// TWENY.RC
...
#include <os2.h>
#include "TWENY.H"
...
STRINGTABLE
{
    ST_ERR1, "Error Message"
}
...
```

This way of writing the resources offers two advantages:

- It makes it easier to maintain the whole application
- It makes the IDs accessible to other modules (to the source code, for instance)

As we have seen for the module definition file (Chapter 2), the resources inserted in the RC file are characterized by the load options and the memory management options that are used to optimize the actions of the OS/2 Memory Manager. These options are listed to the right of the STRINGTABLE directive, according to the following syntax:

```
STRINGTABLE [load options][memory management options]
```

For the load options, you can choose between PRELOAD (which is the default value) and LOADONCALL. The memory management options are MOVEABLE, DISCARDABLE, and FIXED; the first two constitute the default.

## Loading a String

Loading a string into the application from the resource file's STRINGTABLE area is an operation you can perform with the *WinLoadString()* function:

```
#include INCL_WINMESSAGEMGR
LONG APIENTRY WinLoadString(   HAB hab,
                              HMODULE hmod,
                              ULONG id,
                              LONG cchMax,
                              PSZ pchBuffer);
```

<i>Parameter</i>	<i>Description</i>
hab	Handle to the anchor block
hmod	Handle of the module from which the string is to be read
id	ID of the string
cchMax	Maximum length of the buffer that will contain the string
pchBuffer	Pointer to the buffer that will accommodate the string after it has been loaded
<i>Return Value</i>	<i>Description</i>
LONG	Number of characters actually read

*WinLoadString()* returns the number of characters read from the string indicated by the ID given as the third parameter. The text is transferred to the buffer pointed to by *pszBuffer* which is the size of *cchMax* characters. The first and the second parameter indicate, respectively, the handle of the anchor block and the module from which the string is loaded. In most cases, you will refer directly to the executable, by indicating *NULLHANDLE*. An example of the use of *WinLoadString()* will help to clarify the rules that govern its use.

```
CHAR szString[ 30 ] ;
LONG lLen ;
...
lLen = WinLoadString(   hab, NULLHANDLE,
                      ST_ERR1,
                      sizeof( szString), szString ) ;
...
```

The IDs of the strings in a STRINGTABLE, and even those of other resources, must be accessible from within the source code, as is demonstrated by the syntax of *WinLoadString()*. The insertion of the IDs in the application's header file or in some other .H file specifically created for this purpose, is the most efficient way to write a PM application.

To access a resource from some other module, you will have to get its handle by calling the function *WinLoadLibrary()*:

```
#define INCL_WINLOAD
HLIB APIENTRY WinLoadLibrary( HAB hab, PSZ libname ) ;
```



<i>Parameter</i>	<i>Description</i>
hab	Handle of the anchor block
libname	Name of the library to load
<i>Return Value</i>	<i>Description</i>
HLLIB	Handle of the module or NULLHANDLE in case of failure

In addition to the *anchor block* you also need to know the physical name of a DLL on the file system. This function first checks that the indicated DLL is present in memory, and then increments by one its usage count. If the DLL is not loaded, it will be transferred into the system's memory. *WinLoadLibrary()* is inspired by the *DosLoadModule()* function, and has a similar syntax.

## Defining Computed IDs

When you need to insert text strings that are related one to another—like a series of error messages—it is often more practical and convenient to assign a starting ID to the first string, and then specify consecutive values for the following strings. In the header file, you will have only one define—`ST_ERR`, for instance—that will be assigned a value that is easy to deal with:

```
// TWENY.H
...
#define ST_ERR 1000
...
```

In the resource file's `STRINGTABLE` area you will then have several text strings associated with their respective messages that will be displayed if the program incurs some kind of runtime error. The second string will be assigned an ID that is equal to that of the previous message increased by one:

```
// TWENY.RC
...
#include "tweny.h"
...
STRINGTABLE
{
    ST_ERR + 0, "Cannot create the file"
    ST_ERR + 1, "File not found"
    ST_ERR + 2, "Error writing file"
    ...
}
...
```

Here the IDs assigned to the three strings are, respectively 1000, 1001, and 1002. This approach simplifies the definition of IDs in the header file, and also enables you to retrieve the strings during runtime. The only thing to be careful of is selecting a starting ID that is large enough that all other computed IDs don't give rise to conflicts

with some other existing ID. Another way to make sure that IDs are correctly assigned to the various strings is to end the sequence with an empty string:

```
...
STRINGTABLE
{
    ST_ERR + 0, "Cannot create the file"
    ST_ERR + 1, "File not found"
    ST_ERR + 2, "Error writing file"
    ...
    ST_ERR + n, ""
}
...
```

This solution is extremely convenient when you need to insert several text strings automatically and repeatedly in a listbox or in a combobox. Here's how to go about it. The strings will always be loaded one string at a time, through *WinLoadString()*. This function call is, however, inserted in a *while* loop that will test the value returned by *WinLoadString()*:

```
...
i = ST_ERR ;
...
while( WinLoadString( ..., ..., i++, ..., ...))
{
    ...
}
...
```

The ID of the string to load is contained in the identifier *i*, which is incremented automatically through each execution of the loop. When *i* reaches a nonexistent string or the empty string, *WinLoadString()* will return a null value, and thus terminate the string loading process. In the *while* loop's body you could, for example, insert the retrieved string in a listbox or in a combobox.

## ***Why You Should Use a STRINGTABLE***

The strings listed in a *STRINGTABLE* are stored temporarily or indefinitely in a character array declared in the application's source code. If their final destination is in any case the source code, why bother about defining them in a resource file? The main reason for this is to be found in software localization. This means the process by which an application is translated into some other language. Cleanly separating the source code from the text strings displayed by the application grants a simpler conversion of the output into several languages; at the same time, it will also grant a much higher degree of protection of the programmer's coding efforts.

By doing this you can also think about creating a resource file containing not only the application's strings, but also the menu templates and the dialog templates in different languages. It would be up to the application's installation utility or to the

user to select a the final language desired. However, this solution is practical only for small-to-medium-sized executables, otherwise the overall size of the program would be overwhelming.

## MESSAGETABLE

This directive performs almost the same task as a STRINGTABLE. For this reason it is rarely used in a PM application. The syntax of MESSAGETABLE is the same as that of STRINGTABLE. Each text string can be at most 255 characters and is assigned a unique ID defined by the programmer and within the range from 0 and 64K. The only difference is to be found when the strings are loaded into the application. The strings of a MESSAGETABLE resource are loaded into the application via a call to *DosGetResource()*:

```
#define INCL_DOSRESOURCES
APIRET APIENTRY DosGetResource(  HMODULE hmod,
                                ULONG idType,
                                ULONG idName,
                                PPVOID ppb) ;
```

<i>Parameter</i>	<i>Description</i>
hmod	Handle of the module from which the string is to be loaded
idType	ID of the resource type
idName	ID of the resource
ppb	Pointer to a 32 bit identifier holding the offset of the resource
<i>Return Value</i>	<i>Description</i>
APIRET	Error code or zero if the indicated resource was actually loaded

This function will automatically allocate a memory page and load the resources into it. The type of resource is indicated by one of the defines listed in Table 5.6.

The fourth parameter is the address of a 32-bit identifier holding the address of the resource. With *DosQueryResourceSize()* you can get to the physical size of the resource:

```
#define INCL_DOSRESOURCES
APIRET APIENTRY DosQueryResourceSize( HMODULE hmod,
                                       ULONG idt,
                                       ULONG idn,
                                       PULONG pulsize) ;
```

<i>Parameter</i>	<i>Description</i>
hmod	Handle of the module from which the resource is to be loaded
idt	ID of the resource type
idn	ID of the resource
pulsize	Address of a ULONG holding the size of the resource
<i>Return Value</i>	<i>Description</i>
APIRET	Error code or zero if the operation was successful

**Table 5.6 Types of Resources That Can Be Retrieved with *DosGetResource()*; for a MESSAGE TABLE You Need to Select RT\_MESSAGE.**

<i>Type of Resource</i>	<i>Value</i>	<i>Description</i>
RT_POINTER	01	Mouse pointer's shape
RT_BITMAP	02	Bitmap
RT_MENU	03	Menu template
RT_DIALOG	04	Dialog template
RT_STRING	05	String table
RT_FONDIR	06	Font directory
RT_FONT	07	Font
RT_ACCELTABLE	08	Accelerator table
RT_RC DATA	09	Binary data
RT_MESSAGE	10	Error messages' table
RT_DLGINCLUDE	11	Dialog include file
RT_VKEYTBL	12	Virtual key table
RT_KEYTBL	13	Key to UGL tables
RT_CHARTBL	14	Glyph-to-character tables
RT_DISPLAYINFO	15	Display information
RT_FKASHORT	16	Function key area short form
RT_FKALONG	17	Primary or secondary page of scan codes
RT_HELPTABLE	18	Help table resource
RT_HELPSTABLE	19	Resource that can be used as help
RT_FDDIR	20	DBCS uniq/font driver directory
RT_FD	21	DBCS uniq/font driver
RT_MAX	22	1st unused Resource Type

Once you're finished using a particular resource, it is good practice to release the memory associated with it by calling *DosFreeResource()*:

```
#define INCL_DOSRESOURCES
APIRET APIENTRY DosFreeResource( PVOID pb );
```

<i>Parameter</i>	<i>Description</i>
pb	Address of the resource

<i>Return Value</i>	<i>Description</i>
APIRET	Error code or zero if the operation was successful

The only parameter required by this function is the pointer previously produced by *DosGetResource()*. Furthermore, the API of PM also provides the function *WinLoadMessage()*, which is much easier to use than the previous three:

```
#define INCL_WINMESSAGEGR
LONG APIENTRY WinLoadMessage(  HAB hab,
                               HMODULE hmod,
                               ULONG id,
                               LONG cchMax,
                               PSZ pchBuffer);
```

<i>Parameter</i>	<i>Description</i>
hab	Handle of the anchor block
hmod	Handle of the module from which the resource is to be loaded
id	ID of the string
cchMax	Maximum length of the buffer containing the string
pchBuffer	Buffer that will accommodate the string once it has been loaded
<i>Return Value</i>	<i>Description</i>
LONG	Number of characters actually read

The syntax of this function is identical to that of *WinLoadString()*; therefore it is generally preferable to resort to a STRINGTABLE and to *WinLoadString()*.

## WINDOWTEMPLATE

The problem of creating a window through *WinCreateStdWindow()* and *WinCreateWindow()* has been covered in previous chapters. The parameters that need to be given when calling one of these functions will cause the creation of one or more windows, featuring different style attributes. If you need to create multiple correlated windows (like several child windows inside one parent window that simply acts as the principal pixel provider by means of its client area) then you need to call *WinCreateStdWindow()* and/or *WinCreateWindow()* repeatedly.

An alternative solution is offered by the WINDOWTEMPLATE directive in a resource file. As this directive's name suggests, WINDOWTEMPLATE defines the overall structure of a window—simply, the description of a set of windows. The syntax of WINDOWTEMPLATE is similar to that of the text resources already seen and those that will come in the next chapters dedicated to menus and dialog windows.

```
WINDOWTEMPLATE window-ID [load options][memory management options]
{
    ...
}
```

The parameter *window-ID* refers to a numeric ID used to distinguish each WINDOWTEMPLATE present in the resource file. The value of the ID is an integer within the range 0 to 65,536. The load options and the memory management options are always, respectively, LOADONCALL or PRELOAD; and MOVEABLE, DISCARDABLE, and FIXED. The WINDOWTEMPLATE block contains the information needed to create a window. The objective is that of generating a window frame equipped with all the traditional controls completed by a client area. Like any generic window produced with *WinCreateStdWindow()*, you need both a frame window and a client window. At the

resource file level, this is the direct consequence of the FRAME and WINDOW directives inside the WINDOWTEMPLATE block.

FRAME, as you have guessed, instructs the resource compiler to generate all information necessary for creating a frame window. The syntax of FRAME is:

```
FRAME title, ID, x, y, cx, cy, FS_ | WS_, FCF_, data
```

Figure 5.9 compares the syntax of the FRAME directive to that of the *WinCreateWindow()* function used for creating a frame windows.

The number assigned to each item with the FRAME directive recalls its corresponding parameter in *WinCreateWindow()*. In both cases, the syntax is identical. The FRAME directive is not responsible for defining the owner or the parent of the frame window because this information is not available when a window template is written. It will be specified when the window template gets loaded at run time. Furthermore, in the FRAME directive there is no item for indicating what values should be assigned to the

```

1 FRAME
2   "MyFrame",
3   ID_FRAME,
4   10,
5   20,
6   150,
7   90,
8   WS_VISIBLE | WS_CLIPCHILDREN,
9   FCF_SYSMENU | FCF_TITLEBAR | FCF_SIZEBORDER |
   FCF_MINMAX | FCF_SHELLPOSITION | FCF_ICON |
   FCF_ACCELTABLE,
10  NULL

WinCreateWindow(
                                HWND_DESKTOP,
1   WC_FRAME,
2   "MyFrame",
8   WS_VISIBLE,
4   10,
5   20,
6   150,
7   90,
                                NULL,
                                HWND_TOP,
3   ID_FRAME,
9   &myFCF,
10  NULL) ;

```

**Figure 5.9** Comparison of the syntax of the FRAME directive for the resource compiler with that of the *WinCreateWindow()* function in the source code.

presentation parameters. These will be sent to the resource compiler by means of the appropriate `PRESPARAMS` directive. Therefore, creating a frame window with the `FRAME` directive or by calling the `WinCreateWindow()` will produce the same result, provided the starting data are the same.

The “missing” items in the `FRAME` directive’s syntax compared to the parameters of `WinCreateWindow()` reflect the operational differences of the two solutions. `FRAME` is not a `_System`-type function, where all parameters must always be present even if they are useless in a specific case. With the `FRAME` directive, the values that are not needed are simply left out, and the corresponding parameter is not written. In Figure 5.9, it would be nonsense to declare a `NULL` as the last item; it is sufficient to specify nothing at all (the presence of `NULL` in Figure 5.9 serves only the purpose of demonstrating the similarities between the two syntaxes).

The `WINDOWTEMPLATE` resource, enriched by the `FRAME` directive, now becomes:

```
WINDOWTEMPLATE ID_WINDTEMP
{
    FRAME "Resource window", ID_WINDTEMP, 50, 50, 140, 80,
        WS_VISIBLE,
        FCF_MINMAX | FCF_SIZEBORDER | FCF_SYSMENU | FCF_TITLEBAR
    {
        ...
    }
}
```

Even in `FRAME` there exists the directive `FCF_`, which governs the presence of controls. The client window is missing, as it should be. The `WINDOW` directive defines the structure of a generic window according to a syntax that is very close to that of `FRAME`:

```
WINDOW title, ID, x, y, cx, cy, class, FS_ | WS_, FCF_, dati
```

In comparison to `FRAME`, there is just one more item—`class`—that defines the class to which the window belongs. As you might expect, with this parameter it is possible to specify the name of one of the predefined PM classes, and therefore possible to create a frame window, or even more interesting, a window of some class already registered in the application.

```
WINDOWTEMPLATE ID_WINDTEMP
{
    FRAME "Resource window", ID_WINDTEMP, 50, 50, 140, 80,
        WS_VISIBLE,
        FCF_MINMAX | FCF_SIZEBORDER | FCF_SYSMENU | FCF_TITLEBAR
    {
        WINDOW "", FID_CLIENT, 0, 0, 0, 0, "myclass",
            WS_VISIBLE | WS_CLIPCHILDREN
    }
}
```

The parenthood relationship between the window described through `FRAME` and the window defined with `WINDOW` can be inferred from the block layout of the template. In the layout of the client window, you will have the `FID_CLIENT` ID that is used by default

by each frame window to identify its client. This requires an explicit or indirect reference to PMWIN.H among the header files of the resource file. The size of the client window is computed automatically on the basis of the frame window's size at the moment it is created, taking into consideration the control windows created by the FCF\_ flags.

The window template's structure is not limited to the two levels of a FRAME and WINDOW, as shown in the previous example. If the design needs more child windows for the client window, you can insert a new subordinate block under the WINDOW directive, using a pair of braces.

```
WINDOWTEMPLATE ID_WINDTEMP
{
    FRAME "Resource window", ID_WINDTEMP, 50, 50, 140, 80,
        WS_VISIBLE,
        FCF_MINMAX | FCF_SIZEBORDER | FCF_SYSMENU | FCF_TITLEBAR
    {
        WINDOW "", FID_CLIENT, 0, 0, 0, 0, "myclass",
            WS_VISIBLE | WS_CLIPCHILDREN
        {
            ...
        }
    }
}
```

Once this phase is over, and once you have defined in the header file all the IDs that are to be used, there is nothing else left other than changing the source code to load the WINDOWTEMPLATE during the application's run time. To perform this operation, call the *WinLoadDlg()* function, rather than the usual *WinCreateStdWindow()* function.

```
#define INCL_WINDIALOGS
HWND APIENTRY WinLoadDlg(    HWND hwndParent,
                             HWND hwndOwner,
                             PFNWP pfnDlgProc,
                             HMODULE hmod,
                             ULONG idDlg,
                             PVOID pCreateParams) ;
```

<i>Parameter</i>	<i>Description</i>
hwndParent	Handle of the parent window of the frame described in the window template
hwndOwner	Handle of the owner window of the frame described in the window template
pfnDlgProc	Address of the function that plays the role of dialog procedure
hmod	Handle of the module from which the window template is to be loaded
idDlg	ID of the window template



pCreateParams	Address of an area containing the window's presentation parameters
<b>Return Value</b>	<b>Description</b>
HWND	Handle of the created window, generally the frame window

This function is used mainly for creating a *modeless* dialog window. This is simply an extension of WINDOWTEMPLATE's normal use, and a practical demonstration of the high degree of flexibility of PM's API. The description of the syntax of *WinLoadDlg()* refers to this particular situation.

The first two parameters of *WinLoadDlg()* identify the parent and the owner window. When, as in this example, the WINDOWTEMPLATE describes a solution equivalent to what can be obtained through *WinCreateStdWindow()*, the only viable solution is—in both cases—to specify HWND\_DESKTOP. The third parameter identifies the function that acts as what is known as the *dialog procedure* of a modeless dialog window (Chapter 8). In this case, it specifies NULL because both the frame and the client have window procedures of their own.

The handle to the module is set to NULLHANDLE when the relevant information is available in the same executable as the application. The fifth parameter corresponds to the ID assigned to the WINDOWTEMPLATE in the resource file: ID\_WINTEMP. The pointer pCreateParams is not used, and thus is given the value of NULL. If other presentation parameters were necessary, it would be convenient to name them directly in the resource file. The return value of *WinLoadDlg()* corresponds to the handle of the frame window. Different from *WinCreateStdWindow()*, you do not know the handle of the client window, which is easy to get to with *WinWindowFromID()*.

*WinLoadDlg()* is a macro function that calls other two API functions of PM to produce several windows simultaneously. From this point of view, it is similar to *WinCreateStdWindow()*. The main difference between these two functions is to be found in the source of the information: the resource file vs. the application's source code.

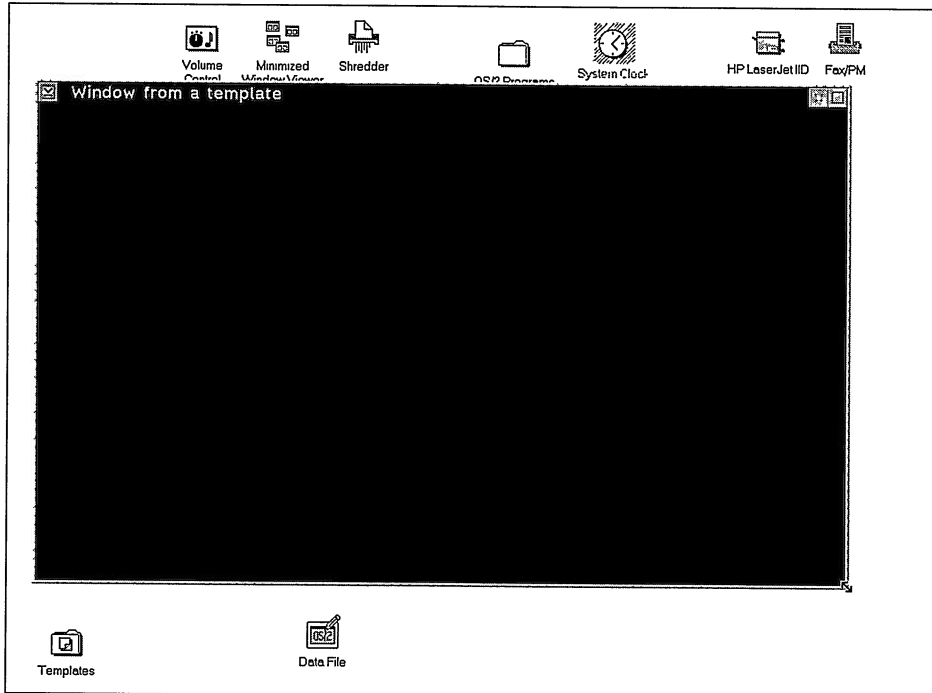
---

## Using Resources



After all this theory, it is time to enrich an OS/2 application with a resource file and a header file. The .RC file contains a STRINGTABLE area with the name of a class to be registered, and a WINDOWTEMPLATE resource that takes the place of a call to *WinCreateStdWindow()*. The *main()* now looks different compared to the previous listings, mainly because of the call to *WinLoadDlg()*. Listing 5.4 shows all the possible variations, while Figure 5.10 shows how the application looks during run time. Visually, there are no great differences with respect to the windows produced in the previous examples with *WinCreateStdWindow()*.

In the WINDOWTEMPLATE directive of Listing 5.4, the FCF\_SHELLPOSITION flag delegates to PM the task of positioning and sizing the frame window; the values given in the template are thus ignored. The presence of this flag also allows the WM\_PAINT message to be recognized in the window procedure of the class registered



**Figure 5.10** A PM application characterized by a window produced through a description in a resource file and through a WINDOWTEMPLATE directive.

by the code, and therefore to paint the client window in red. If you were to leave out the FCF\_SHELLPOSITION flag, the WM\_PAINT message would not be received, and the client window would not be colored. However the WM\_ERASEBACKGROUND message would still arrive, and it contains in mp1 the handle of a presentation space and in mp2 a pointer to a RECTL structure describing the screen portion that has been invalidated. If you want to do without the FCF\_SHELLPOSITION flag in order to specify the position and the size of the frame window directly in the window template, then you must intercept the WM\_ERASEBACKGROUND and take advantage of the information contained in mp1 and mp2. This is a code fragment demonstrating the technique:

```

...
case WM_ERASEBACKGROUND:
    WinFillRect( (HPS)LONGFROMMP( mp1),
                (PRECTL)PVOIDFROMMP( mp2),
                CLR_RED) ;
    return (MRESULT)TRUE ;
...

```

Be careful, however. The contents of mp1 are a handle to the *frame window's* presentation space, and not that of the client window. *WinFillRect()* will thus refer to the space of the frame, which will later be covered by the client.

**Table 5.7 Directives That Introduce Binary Resources in a RC File**

<i>Directive</i>	<i>Description</i>
BITMAP	Image
FONT	Set of characters and symbols
ICON	Icon for displaying a minimized window
POINTER	Mouse cursor on the screen
RESOURCE	Definition of some custom resource by the application

It is important that the name of the class of the window declared with the `WINDOW` directive in the `WINDOWTEMPLATE` be typed exactly, even as to upper- and lowercase, as in the source code. To avoid typing errors, it is good to specify the name of the class of the client window directly in the application's header file, and to refer to this define both in the `WINDOWTEMPLATE` as well as in the source code for registering the window's class.

## Binary Resources

Resource files can also accommodate directives that refer to files containing binary resources—typically images. These directives are listed in Table 5.7.

The directives `BITMAP`, `ICON`, `POINTER`, and `FONT` (Table 5.8) all have the same syntax, characterized by the presence of an ID, loading options, memory management options, and the name of the file containing the resource. The directive `RESOURCE` has yet another item, the ID that identifies some kind of new resource.

`ICON` and `POINTER` are both fixed-size bitmaps; their actual size can be determined with `WinQuerySysValue()`. The mouse pointer is a colored image that can be chosen

**Table 5.8 Syntax for Indicating Binary Resources in a Resource File**

<i>Binary Resources</i>	<i>Syntax</i>
BITMAP	<code>BITMAP bitmapID [loading options] [memory management options] filename</code>
ICON	<code>ICON iconID [loading options] [memory management options] filename</code>
POINTER	<code>POINTER pointerID [loading options] [memory management options] filename</code>
FONT	<code>FONT fontID [loading options] [memory management options] filename</code>
RESOURCE	<code>RESOURCE typeID resourceID [loading options] [memory management options] filename</code>

from those available in PM or defined by the programmer. By default, the mouse cursor is a black arrow pointing northwest (define *SPTR\_ARROW*).

The term *icon* has different meanings in GUI environments. In Microsoft Windows, for example, an icon almost invariably indicates some minimized window, or references some executable (as in the Program Manager). In PM, instead, an icon can represent objects of WPS; in other words, it can represent physical devices, data files, programs, and folders. In a resource file, the *ICON* directive can be used to identify what image should be displayed in the title bar and what image should represent the application in WPS. Often, even the documents generated by a program are represented in WPS by specific icons, which are usually different from those of the application, although they normally look similar to one. By examining the *Templates* folder of a system equipped with some personal productivity application, like Fax/PM, you can see this type of icon (Figure 5.11).

## Loading an Icon

In the resource file there is information regarding the icon to associate with a window when it is to be displayed minimized in a folder. This aspect of programming has

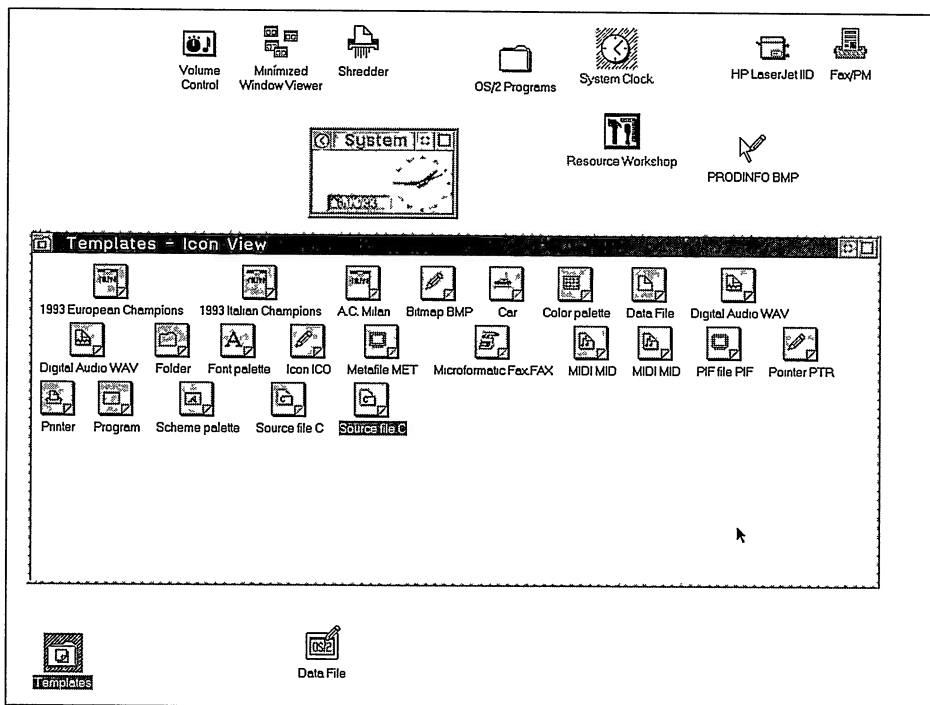


Figure 5.11 The Templates folder containing some object generators of Fax/PM for OS/2 2.1.

become less important because it is preferable to hide a window, rather than minimize it following the behavioral rules of WPS. The first operation to perform is to draw an icon. The tool used for this is ICONEDIT.EXE, which is provided in the Toolkit of PM. In OS/2 2.1, the icons are colored and device independent (Figure 5.12).

Once an image has been saved in a file, assign the icon an ID directly in the resource file, according to the syntax:

```
...
// TWENY.RC
...
#include "tweny.h"
...
ICON RS_ICON TWENY.ICO
...
```

In the header file you will find the define of the RS\_ICON ID, which is used as the eighth parameter of *WinCreateStdWindow()*; in this way you can be sure that you are using that image as the title bar icon (the presence of which is implied by the flags FCF\_SYSMENU and FCF\_ICON). The call to *WinCreateStdWindow()* will also cause the

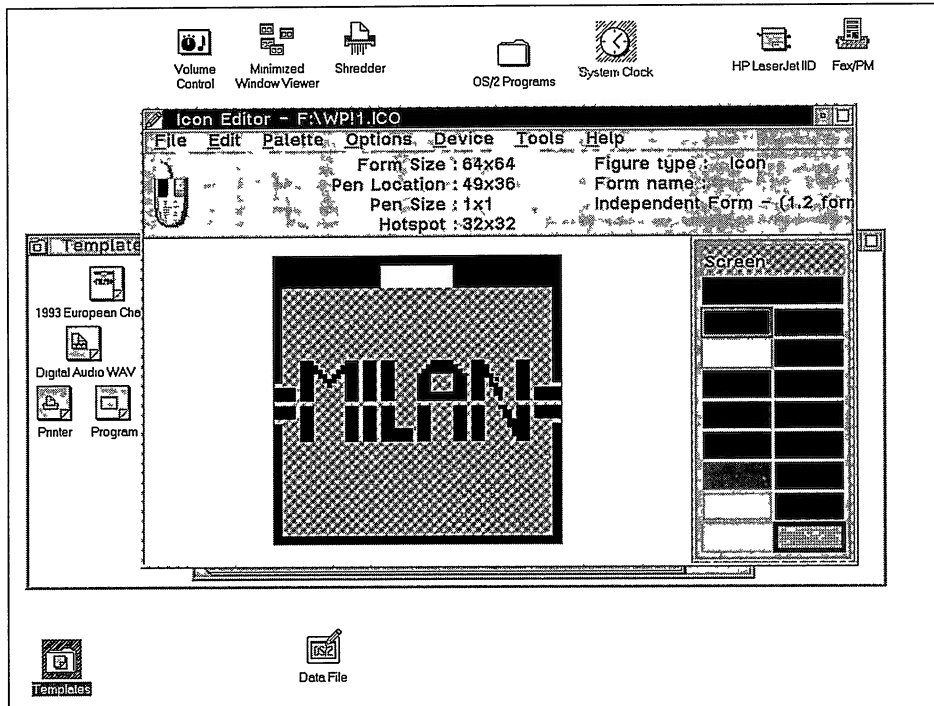


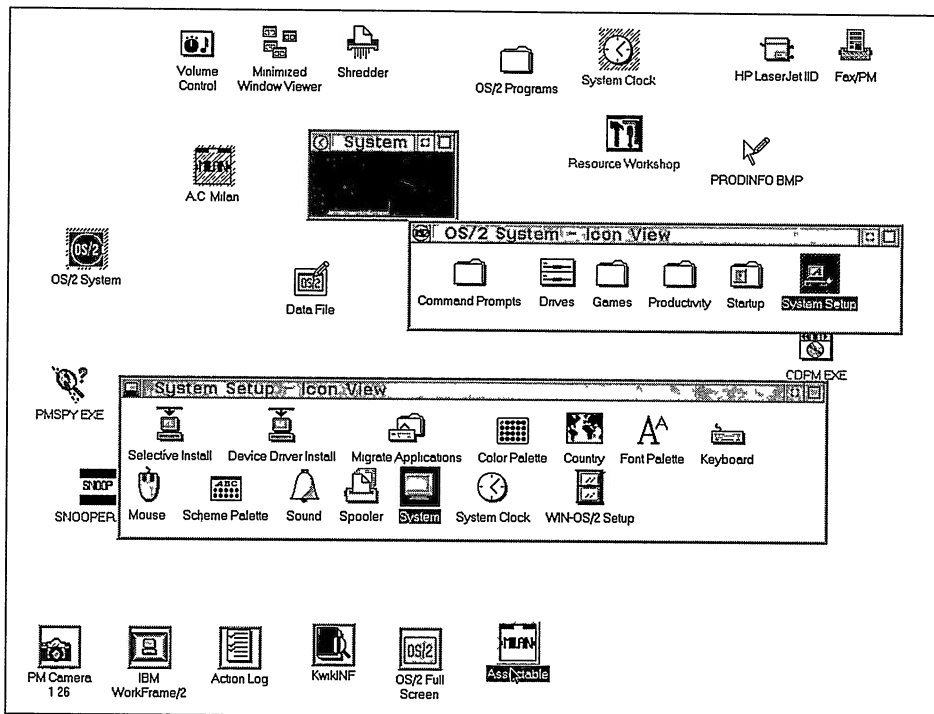
Figure 5.12 ICONEDIT.EXE is the tool that lets you draw icons, cursors, and bitmaps for OS/2 PM.



association of the icon present in the resource file with the application window. To achieve this, you have to set the `FS_ICON` or the `FCF_ICON` flag when the window is being created. Figure 5.13 shows the application of Listing 5.5, equipped with a custom title bar icon.

The resource ID in `WinCreateStdWindow()` refers not only to the title bar icon, but also to a menu and an accelerator table. The ID that is assigned to these three resources must be unique. For this reason a generic name such as `RS_ALL` is used. A convention used in this book is that of defining all binary resources with the `RS_` prefix and indicating the kind of the resource after the underscore. When an application needs to associate a main window with the three possible resources, the header file will look like this:

```
// TWENY.H
...
#define RS_ALL      300
#define RS_ICON     RS_ALL
#define RS_MENU     RS_ALL
#define RS_ACCELTABLE RS_ALL
...
```



**Figure 5.13** The icon specified in the resource file is shown on the screen when the application is minimized.

In the call to *WinCreateStdWindow()*, the define *RS\_ALL* is indicated when at least two resources are needed, while in the RC file every resource will have a specific ID.

The same icon that replaces the standard image in the title bar can actually be used in the application like an object to be displayed in the client window. An icon is just a small bitmap, and therefore can be part of the output produced by an application. The loading of an icon is performed with *WinLoadPointer()*.

```
HPOINTER WINAPI WinLoadPointer(   HWND hwndDesktop,
                                  HMODULE hmod,
                                  ULONG idres) ;
```

<i>Parameter</i>	<i>Description</i>
<i>hwndDesktop</i>	Handle of the desktop window
<i>hmod</i>	Handle of the module from which the resource is to be loaded
<i>idres</i>	Resource ID
<i>Return Value</i>	<i>Description</i>
HPOINTER	Handle to the loaded pointer or NULLHANDLE in case of error

Once you have the handle of the icon, you can display the icon with *WinDrawBitmap()* or *WinDrawPointer()*.

---

## Predefined Icons and Pointers

PM comes with a set of predefined icons and pointers that can be used in a variety of situations: to flag operations or messages, drag & drop operations, and more. These predefined resources are characterized by an ID with the *SBMP\_* or the *SPTR\_* prefix. To know what the predefined bitmaps are, just look at the contents of *PMWIN.H*, and search for the following *SBMP\_* defines (Table 5.9).

**Table 5.9 Defines for the Predefined Bitmaps in PM**

<i>Bitmap</i>	<i>Value</i>	<i>Description</i>
<i>SBMP_OLD_SYSMENU</i>	01	Old system menu image.
<i>SBMP_OLD_SBUPARROW</i>	02	Old image for arrow pointing up.
<i>SBMP_OLD_SBDNARROW</i>	03	Old image for arrow pointing down.
<i>SBMP_OLD_SBRGARROW</i>	04	Old image for arrow pointing right.
<i>SBMP_OLD_SBLFARROW</i>	05	Old image for arrow pointing left.
<i>SBMP_MENUCHECK</i>	06	Checkmark for a state setter menu item.
<i>SBMP_OLD_CHECKBOXES</i>	07	Old image for a checkbox.
<i>SBMP_BTNCORNERS</i>	08	Corners of <i>pushbutton</i> .
<i>SBMP_OLD_MINBUTTON</i>	09	Old image of the minimize button.
<i>SBMP_OLD_MAXBUTTON</i>	10	Old image of the maximize button.

(continued)

**Table 5.9 (Continued)**

<i>Bitmap</i>	<i>Value</i>	<i>Description</i>
SBMP_OLD_RESTOREBUTTON	11	Old image of the restore button.
SBMP_OLD_CHILDSYSMENU	12	Old image of a child window's title bar icon.
SBMP_DRIVE	15	Symbol used by the File Manager to represent a drive.
SBMP_FILE	16	Symbol used by the File Manager to represent a file.
SBMP_FOLDER	17	Symbol used by the File Manager to represent a subdirectory containing other directories.
SBMP_TREEMINUS	18	Symbol used in folders in tree display mode.
SBMP_TREEPLUS	19	Symbol used in folders in tree display mode.
SBMP_PROGRAM	22	Icon used by the File Manager to represent an executable program.
SBMP_MENUATTACHED	23	Horizontal arrow used to indicate a second or lower level pulldown menu.
SBMP_SIZEBOX	24	Icon used to identify the box for restoring the original size of a window (typically corresponds to a vertical or horizontal scrollbar).
SBMP_SYSMENU	25	System menu icon.
SBMP_MINBUTTON	26	Minimize icon.
SBMP_MAXBUTTON	27	Maximize icon.
SBMP_RESTOREBUTTON	28	Restore icon.
SBMP_CHILDSYSMENU	29	Child window's system menu icon.
SBMP_SYSMENUDEP	30	Selected title bar icon.
SBMP_MINBUTTONDEP	31	Selected minimize icon.
SBMP_MAXBUTTONDEP	32	Selected maximize icon.
SBMP_RESTOREBUTTONDEP	33	Selected restore icon.
SBMP_CHILDSYSMENUDEP	34	Selected child window's menu bar icon.
SBMP_SBDNARROW	35	Arrow pointing down in a scrollbar.
SBMP_SBDNARROWDEP	36	Selected arrow pointing down in a scrollbar.
SBMP_SBDNARROWDIS	37	Disabled arrow pointing down in a scrollbar.
SBMP_SBLFARROW	38	Arrow pointing left in a scrollbar.
SBMP_SBLFARROWDEP	39	Selected arrow pointing left in a scrollbar.
SBMP_SBLFARROWDIS	40	Disabled arrow pointing left in a scrollbar.
SBMP_SBRGARROW	41	Arrow pointing right in a scrollbar.
SBMP_SBRGARROWDEP	42	Selected arrow pointing right in a scrollbar.
SBMP_SBRGARROWDIS	43	Disabled arrow pointing right in a scrollbar.

*(continued)*



Table 5.9 (Continued)

<i>Bitmap</i>	<i>Value</i>	<i>Description</i>
SBMP_SBUPARROW	44	Arrow pointing up in a scrollbar.
SBMP_SBUPARROWDEP	45	Selected arrow pointing up in a scrollbar.
SBMP_SBUPARROWDIS	46	Disabled arrow pointing up in a scrollbar.
SBMP_COMBODOWN	47	Arrow pointing down in a window of the WC_COMBOBOX class.
SBMP_CHECKBOXES	48	Set of symbols used to represent all states of a checkbox.

Loading these bitmaps is performed with the *WinGetSysBitmap()* function, which is very simple and easy to use:

```
#define INCL_WINPOINTERS
HBITMAP WINAPI WinGetSysBitmap( HWND hwndDesktop, ULONG ibm );
```

<i>Parameter</i>	<i>Description</i>
hwndDesktop	Handle of the desktop, HWND_DESKTOP
ibm	ID of the system bitmap to load

<i>Return Value</i>	<i>Description</i>
HBITMAP	Handle of the loaded system bitmap or NULLHANDLE in case of error

The first parameter must always be the define `HWND_DESKTOP`, while the second one corresponds to any of the `SBMP_` defines described in Table 5.9. The return value of this function is the handle to a predefined bitmap of PM. In practice, *WinGetSysBitmap()* is just a special case of the more generic *GpiLoadBitmap()*, a function that would seem to belong to the Gpi group, but is physically located in `PMWIN.DLL`. Whatever method is used to get a bitmap handle, to draw it on the screen you have to call *WinDrawBitmap()*:

```
#define INCL_WINMESSAGEGR
BOOL WINAPI WinDrawBitmap( HPS hpsDst,
                           HBITMAP hbm,
                           PRECTL pwrSrc,
                           PPOINTL pptlDst,
                           LONG clrFore,
                           LONG clrBack,
                           ULONG fl );
```

<i>Parameter</i>	<i>Description</i>
hpsDst	Handle of the target presentation space
hbm	Handle of the bitmap
pwrSrc	Part of the bitmap to be displayed
pptlDst	Lower left-hand corner of the bitmap on the drawing surface

<i>Parameter</i>	<i>Description</i>
clrFore	Foreground color
clrBack	Background color
fl	Display flag
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

*WinDrawBitmap()* is an output function and thus requires *hps* as its first parameter, and then the bitmap's handle. The third parameter is of the type *RECTL* and describes the size of the bitmap to be shown on the screen. You can select the whole bitmap by specifying *NULL*. The *POINTL* structure contains the coordinates of the lower left-hand corner of the bitmap, the point that will be used by the function as the starting point for displaying the image. The next two *LONG* parameters define respectively the foreground and the background color. Their value is ignored if the last parameter is set to *DBM\_IMAGEATTRS*. In all other conditions, the indicated colors are employed (Table 5.10). The return value informs the application about the success or failure of the operation.

To access one of PM's predefined mouse pointers, use the function *WinQuerySysPointer()*:

```
#define INCL_WINPOINTERS
HPOINTER APIENTRY WinQuerySysPointer( HWND hwndDesktop,
                                       LONG iptr,
                                       BOOL fLoad) ;
```

<i>Parameter</i>	<i>Description</i>
hwndDesktop	Handle to the desktop, <i>HWND_DESKTOP</i>
iptr	ID of a predefined pointer image
fLoad	<i>TRUE</i> requests the creation of a copy of the system bitmap, <i>FALSE</i> accesses the original bitmap

**Table 5.10 Display Options for Bitmaps with the *WinDrawBitmap()* Function**

<i>Flag</i>	<i>Value</i>	<i>Meaning</i>
<i>DBM_NORMAL</i>	0x0000	The bitmap is displayed normally using <i>ROP_SRCCOPY</i> .
<i>DBM_INVERT</i>	0x0001	The bitmap's colors are inverted according to the rules of <i>ROP_NOTSRCCOPY</i> .
<i>DBM_HALFTONE</i>	0x0002	Displays the bitmap after applying to it the <i>OR</i> operator with a pattern of alternating ones and zeros.
<i>DBM_STRETCH</i>	0x0004	The parameter <i>pptlDest</i> no longer refers to a <i>POINTL</i> structure containing the new size that the bitmap must take on the screen.
<i>DBM_IMAGEATTRS</i>	0x0008	The bitmap is displayed with all its original attributes, and the parameters <i>clrFore</i> and <i>clrBack</i> are ignored.

<i>Return Value</i>	<i>Description</i>
HPOINTER	Handle to the loaded pointer or NULLHANDLE in case of failure

The second parameter corresponds to one of the defines listed in Table 5.11. The first one, SPTR\_APPICON, refers to just one blank icon; the generic solution that should be assigned to a window if the programmer does not provide an icon. If the third parameter, fLoad, is set to TRUE, it will instruct *WinQuerySysPointer()* to load the mouse pointer's icon; if fLoad is set to FALSE the function will only return a handle to the system cursor. In both cases, *WinQuerySysPointer()* returns a handle to the system cursor.

To display any of PM's predefined bitmaps, you have to use the *WinDrawPointer()* function, passing it the handle returned by *WinQuerySysPointer()*:

```
#define INCL_WINPOINTERS
BOOL WINAPI WinDrawPointer( HPS hps,
                           LONG x,
                           LONG y,
                           HPOINTER hptr,
                           ULONG fl );
```

**Table 5.11 Identifiers of PM's Predefined Bitmaps That Can Be Accessed Through the Function *WinQuerySysPointer()***

<i>Pointer</i>	<i>Value</i>	<i>Description</i>
SPTR_ARROW	1	Traditional arrow
SPTR_TEXT	2	I beam
SPTR_WAIT	3	Hourglass icon to indicate a wait
SPTR_MOVE	4	Movement of a window
SPTR_SIZENWSE	5	Arrow pointing in the direction Northwest - Southeast
SPTR_SIZENESW	6	Arrow pointing in the direction Northeast - Southeast
SPTR_SIZEWE	7	Double-headed horizontal arrow
SPTR_SIZENS	8	Double-headed vertical arrow
SPTR_APPICON	9	Blank icon
SPTR_ICONINFORMATION	10	Information request
SPTR_ICONQUESTION	11	Question mark
SPTR_ICONERROR	12	STOP sign
SPTR_ICONWARNING	13	Warning icon
SPTR_ILLEGAL	14	Exclamation mark
SPTR_ILLEGAL	18	Forbidden symbol
SPTR_FILE	19	File symbol
SPTR_FOLDER	20	Directory symbol
SPTR_MULTIFILE	21	Multiple stacked files
SPTR_PROGRAM	22	Symbol for executable modules

<i>Parameter</i>	<i>Description</i>
hps	Handle to the presentation space
x	Coordinate of the lower left-hand side corner of the image on the X axis
y	Coordinate of the lower left-hand side corner of the image on the Y axis
hptr	Handle to the pointer to be displayed
fl	Display flag
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The first parameter refers to the handle of the presentation space inside which the mouse pointer defined by the handle *hptr* will be appear. The point of coordinates *x*, *y*—expressed in screen units—defines the lower left-hand corner of the bitmap. The fifth parameter, *fl*, is a flag that tells *WinDrawPointer()* how it should display the bitmap. The values that *fl* can take are summarized in Table 5.12.

## Displaying Predefined Bitmaps

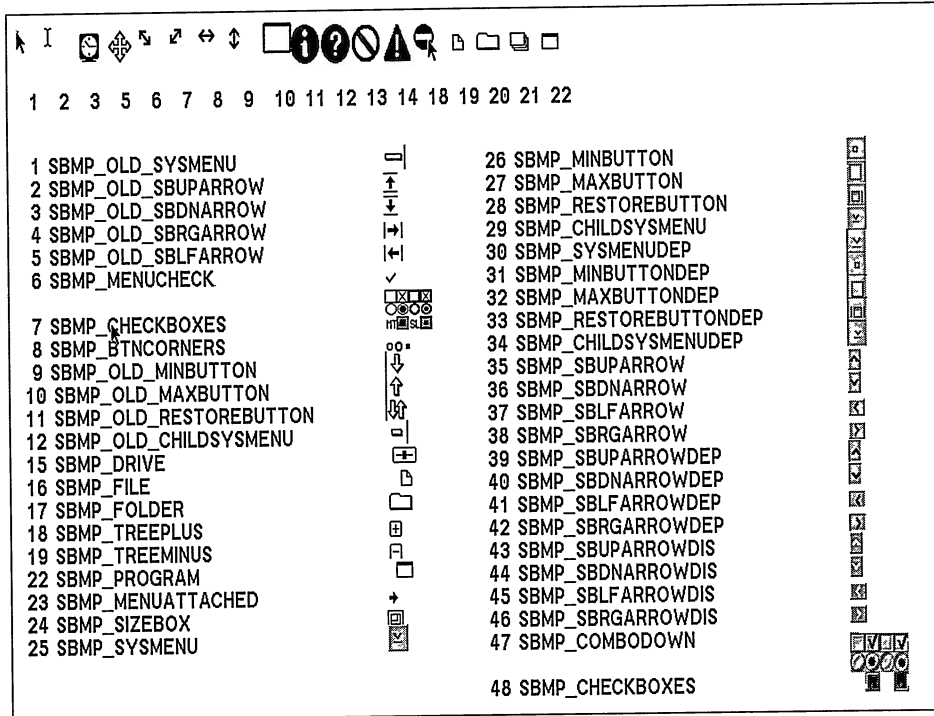
The four functions *WinGetSysBitmap()*, *WinDrawBitmap()*, *WinQuerySysPointer()*, and *WinDrawPointer()* let you retrieve and display PM's predefined bitmaps, either for the structural elements of a window or for the mouse pointer's icon. The bitmaps that can be retrieved with *WinGetSysBitmap()* are characterized by an ID from 1 to 47, with some exceptions. The mouse pointers will have IDs between 1 and 22, and in this case there are also a few exceptions. The following simple application (Listing 5.6) will allow you to display all of PM's predefined bitmaps. Figure 5.14 presents the application's output.



If you wish to display all bitmaps simultaneously on the screen without having to implement any kind of scrolling, the window will have to appear maximized on the screen. There is a flag, *WS\_MAXIMIZE*, but it will not produce any effect if it is specified directly in the *WinCreateStdWindow()* function. The window is displayed without setting the *WS\_VISIBLE* flag; then the window is sized and displayed with the *WinSetWindowPos()* function, where the flags *SWP\_MAXIMIZE*, *SWP\_SHOW*, and *SWP\_ACTIVATE* are all set.

**Table 5.12 Pointer's Display Flags That Can Be Used with *WinDrawPointer()***

<i>Flag</i>	<i>Value</i>	<i>Description</i>
DP_NORMAL	0x0000	The bitmap is displayed exactly as it appears
DP_HALFTONED	0x0001	The bitmap is displayed in halftones
DP_INVERTED	0x0002	The bitmap is displayed inverted



**Figure 5.14** The predefined bitmaps of OS/2 PM, declared in PMWIN.H with the defines SBMP\_ and SPTR\_.

The application described in Listing 5.6 presents yet another peculiarity: The only frame control flag that is specified is FCF\_TASKLIST. In fact, the pixels of the title bar serve to display all relevant facts on VGA resolution. This also means that you have to give up the system menu and the maximize and minimize icons. In order to terminate the application, a case branch is introduced to intercept the clicking of the left mouse button, to which the application will respond by posting the message WM\_QUIT.

The program's logic concentrates on processing the message WM\_PAINT, and it will call the functions *ShowPointers()* and *ShowIcons()* defined in the code. Displaying the mouse's icon works by first retrieving a handle to each of the bitmaps through the *WinQuerySysPointer()* function, and then using this handle in a call to *WinDrawPointer()*:

```
...
hptr = WinQuerySysPointer( HWND_DESKTOP, j, FALSE );

if( !hptr)
    continue ;

WinDrawPointer( hps, x, y, hptr, DP_NORMAL ) ;

...
```

The test performed on the `hptr` handle allows the program to detect the icons in the range from 1 to 22 with which there is no associated bitmap. The coordinates `x`, `y` are calculated in such a way as to display all bitmaps in one horizontal row in the upper half of the screen.

The operation for displaying the system bitmaps is somewhat more complex. The functions to be called are `WinLoadString()`, `WinGetSysBitmap()`, and `WinDrawBitmap()`. The first function is used to retrieve from the resource file's `STRINGTABLE` area the name of a system bitmap as it is described in `PMWIN.H`. `WinGetSysBitmap()` returns a handle to the system bitmap. This handle is then passed to `WinDrawBitmap()`, where the foreground color is set to `CLR_WHITE` and the background color is set to `CLR_BLACK`. The display flag is set to `DBM_NORMAL`.

```

...
for( i = 1; i < 48; i++)
{
    if( !WinLoadString(  HAB( hwnd),
                        0,
                        i,
                        sizeof( buffer),
                        buffer))
        continue ;

    if( i == 7)
        k++ ;

    ...

    hbm = WinGetSysBitmap(  HWND_DESKTOP, i ) ;
    WinDrawBitmap(  hps,
                   hbm,
                   NULL,
                   &pt,
                   CLR_BLACK,
                   CLR_WHITE,
                   DBM_NORMAL) ;

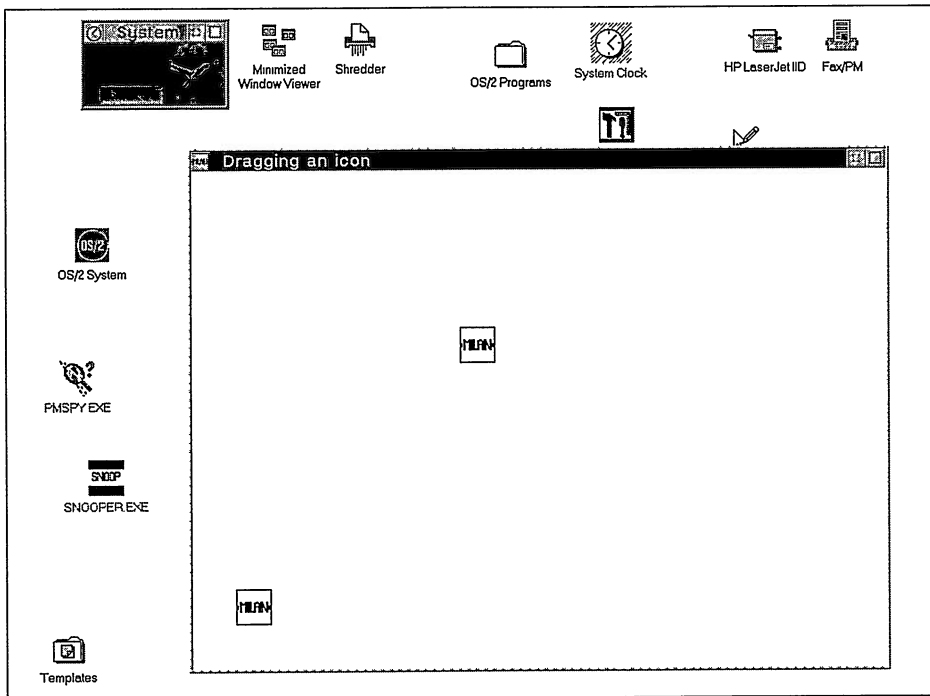
    ...

```

## Moving an Icon in a Window

In OS/2 PM, multiple actions can be performed by *dragging & dropping* various kind of icons. This is also true for WPS objects inside a generic folder. The copying or moving of a file from one directory to another requires you to select a graphical object—an icon—and then to drag it into some other position on the screen. You do this by keeping the right mouse button (2) depressed.

What we have learned about displaying PM's predefined bitmaps is extremely useful for designing an application that can resort to icons for performing practical operations. Figure 5.15 shows the output of an application that displays an icon in its client window and allows you to drag it around with the mouse.



**Figure 5.15** Dragging a graphical object in a PM application by taking advantage of the cursor's functionality.

Before examining the sample code, let's consider the possible solutions. There are three hypotheses that can be considered:

- Replacing the mouse's icon with that of the object to be dragged
- Simulating a complete drag & drop operation
- Implementing a true OS/2 2.1 drag & drop operation, taking advantage of the appropriate API functions

Of these three ideas, the third one is the one that should be implemented. A complete examination of true drag & drop technique appears in Chapter 12. The first solution seems simple to implement and offers, in practical terms, acceptable results. It is an alternative that can be interesting to explore. The second is based on constantly redrawing the dragged object's icon wherever the mouse is currently located. The overall outcome is disconcerting because of flickering that is difficult to get rid of. So, in practice, taking advantage of the *Drg* API of PM is the way to go to implement a professional looking drag & drop operation.



If you replace the mouse's icon with that of the object's icon to be moved, the whole chore is assigned to PM, thus preventing further intervention. The only negative part of this is the disappearance of the mouse's traditional arrow pointer. In Listing 5.7 you can see the application's source code.

The icon is loaded as soon as the WM\_CREATE message is received. At that moment, the overall on-screen size of icon is computed, and the coordinates are stored in a static RECTL structure. The position of the lower left-hand corner is stored in a POINTL structure that will be used later for displaying the icon while processing the WM\_PAINT message.

```

...
case WM_CREATE:
{
    // loading the pointer
    hptr = WinLoadPointer( HWND_DESKTOP, NULLHANDLE, ID_PTR );
    // pointer dimensions
    rc.xRight = SYSVAL( SV_CXPOINTER ) + STARTX ;
    rc.yTop = SYSVAL( SV_CYPOINTER ) + STARTY ;
    rc.xLeft = ptl.x = STARTX ;
    rc.yBottom = ptl.y = STARTY ;

    // icon dimensions
    uscx = (USHORT)SYSVAL( SV_CXPOINTER ) ;
    uscy = (USHORT)SYSVAL( SV_CYPOINTER ) ;
}
break ;
...

...
case WM_PAINT:
{
    HPS hps ;

    hps = WinBeginPaint( hwnd, NULLHANDLE, NULL ) ;
    GpiErase( hps ) ;
    WinDrawPointer( hps, ptl.x, ptl.y, hptr, DP_NORMAL ) ;
    WinEndPaint( hps ) ;
}
break;
...

```

The logic that governs the movement of the icon is concentrated in the processing of the WM\_BUTTON1DOWN and WM\_BUTTON1UP messages. When the left mouse button is pressed inside the client window of the application, you must first check that the cursor's coordinates fall inside the area occupied by the icon on the screen.

```

...
case WM_BUTTON1DOWN:
{
    // check if we are clicking on the icon
    ptl.x = (LONG)SHORT1FROMMP( mp1 ) ;
    ptl.y = (LONG)SHORT2FROMMP( mp1 ) ;

    // skip if not on the icon
    if( !WinPtInRect( HAB( hwnd ), &rc, &ptl ) )
        break ;
    ...
}
...

```



Once this test is passed, the boolean identifier `fDrag` is set to `TRUE`.

```
...
fDrag = TRUE ;
...
```

Then, replace the mouse's arrow pointer icon with the icon corresponding to the image present in the window. To achieve a perfect overlap between the mouse's icon and the image in the client area, it is necessary that the *hot spot* be located exactly in the icon's center. The function `WinSetPointerPos()` requires as its parameters the handle of the desktop window and the new coordinates expressed in screen units. The central point of an icon is easy to find, starting from the lower left-hand side of the rectangle that corresponds to the image's on-screen size. Once that position has been computed, its coordinates are converted into screen units with `WinMapWindowPoints()`.

```
...
// overlapped new mouse pointer to the existing icon
ptl.x = rc.xLeft + uscx / 2 ;
ptl.y = rc.yBottom + uscy / 2 ;
WinMapWindowPoints(hwnd, HWND_DESKTOP, &ptl, 1) ;
WinSetPointerPos(   HWND_DESKTOP,
                   (SHORT)ptl.x,
                   (SHORT)ptl.y) ;
...
```

The display of the new mouse pointer that corresponds to the selected icon is a task performed by the `WinSetPointer()` function.

```
...
// show new pointer
WinSetPointer( HWND_DESKTOP, hptr) ;
break ;
...
```

Once these operations are complete, the mouse pointer looks like the selected icon, and it will look that way as long as it stays within the client window. If it is moved outside the client window, the mouse pointer changes into the standard icon for the underlying window. On the title bar, that is the standard arrow, or over the sizing border it will be the double-headed arrow, and so on. To ensure that if and when the mouse returns to the client window, the selected icon gets restored automatically, you must intercept the `WM_MOUSEMOVE` message. If the mouse movement is not the consequence of selecting the icon—that is, if `fDrag` is `FALSE`—then the default processing is performed by `WinDefWindowProc()`: the window's predefined icon (the traditional arrow) is displayed on the screen. In the other case, the mouse pointer icon is overridden by the previously selected icon. This solution is actually a little redundant—for each `WM_MOUSEMOVE` some time is wasted forcing the repeated appearance of the pointer corresponding to the icon. However, it is a way to ensure that the program works correctly. In fact, dragging the icon is limited to the client window

area. In this case the mouse pointer is represented by the northwest pointing arrow. When it returns to the client window, the cursor again changes into the selected icon. The problem is solved by calling *WinSetPointer()*.

```
...
case WM_MOUSEMOVE:
    if( !fDrag)
        break ;
    WinSetPointer( HWND_DESKTOP, hptr) ;
    return (MRESULT)1L ;
...

```

The final part of the logic governing the movement of an icon is that regarding the *WM\_BUTTON1UP* message. The first operation to carry out is that of checking that the mouse button is truly released after a previous *WM\_BUTTON1DOWN* perform on the icon: You have to test the value of *fDrag*.

```
...
case WM_BUTTON1UP:
    // skip if we are not dragging
    if( !fDrag)
        break ;

    // stop dragging
    fDrag = FALSE ;
...

```

The values contained in the two *SHORT* of *mp1* refer to the coordinates of the mouse on the screen, expressed in client window units. The mouse has been moved on the screen, but its position is unchanged with reference to the icon—it is always at the center of the object. Computing the perimeter of the rectangle occupied by the mouse cursor on the screen is therefore a simple mathematical operation based on the value of *mp1*.

```
...
// calculate new position
ptl.x = rc.xLeft = SHORT1FROMMP( mp1) - uscx / 2 ;
ptl.y = rc.yBottom = SHORT2FROMMP( mp1) - uscy / 2 ;
rc.xRight = uscx + rc.xLeft ;
rc.yTop = uscy + rc.yBottom ;
...

```

Now assign the mouse pointer the typical look of the selection arrow by calling *WinQuerySysPointer()* and *WinSetPointer()*. The mouse's arrow will be positioned exactly at the center of the icon as soon as it is redrawn in its new position, while the icon's bitmap is always fixed in its starting position—the only item that is moved is the mouse's cursor (see Figure 5.15).

```

...
// show arrow pointer
WinSetPointer( HWND_DESKTOP,
               WinQuerySysPointer( HWND_DESKTOP,
                                   SPTR_ARROW, TRUE) );
...

```

The output of the icon is an operation that affects the processing of the WM\_PAINT message. By invalidating the whole client window you can make sure that the icon is displayed at its target on-screen position.

```

...
// show the icon in the client area
WinInvalidateRect( hwnd, NULL, FALSE );
break ;
...

```

Before abandoning the application, you must destroy all objects used by the program, including the icon's handle.

```

...
case WM_DESTROY:
    WinDestroyPointer( hptr ) ;
    break ;
...

```

The technique used in Listing 5.7 can be useful to implement applications that require the user to handle small graphical objects or bitmaps.

---

## Moving a Bitmap in a Window

The technique used for moving an icon can be extended to graphical objects that are a little larger, like bitmaps. However, you will have to face the problem of increased flickering and of the mouse being less responsive to the user's movements.

When you need to move bitmaps that are larger than an icon, a good solution is to draw on the screen an outline rectangle that is the same size as the icon, and then move only the rectangle corresponding to the movements of the mouse. This solution is offered by the system itself, when the user is moving a whole window: The window remains fixed in its starting position on the screen, while the user moves an outline of the window. Only when the user releases the left mouse button will the window actually be transferred to the new position.

Although the logical scheme to follow is simple and linear, when you try to implement it you will face problems due to flickering and to a certain degree of complexity of the code. In fact, the operations are:

- Intercept the pressure of the left mouse button, and decide whether that action was performed in an on-screen rectangle occupied by the bitmap;
- Position the mouse cursor exactly at the center of the bitmap's rectangle;
- Intercept the WM\_MOUSEMOVE message, and decide whether it is subsequent to a previous WM\_BUTTON1DOWN message;
- In processing the WM\_MOUSEMOVE message, you have to display/clear a rectangle that is the same size of the object being moved (the operation can be performed directly in the code dealing with the message or forced by sending a WM\_PAINT message);
- Intercept the message WM\_BUTTON1UP, and decide whether it is subsequent to a previous valid WM\_BUTTON1DOWN, and reposition the object.

These are by no means impossible operations, but to come to our rescue we have PM's *WinTrackRect()* API function, which is designed specifically for this kind of operation:

```
#define INCL_WINTRACKRECT
BOOL APIENTRY WinTrackRect( HWND hwnd, HPS hps, PTRACKINFO pti );
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window where the image is being moved
hps	Handle of the presentation space of the window given by hwnd
pti	Pointer to a TRACKINFO structure
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The first parameter identifies the window inside which the dragging operation is performed. The second one refers to the presentation space of that window, while the third is the address of a TRACKINFO structure.

With *WinTrackInfo()* the application can associate the mouse's movements with the display on the screen of a rectangle the size of which can be set at will to represent the object you want to move. The task performed by this function is rather complex, as is its governing internal logic, even though its syntax is extremely simple. To understand how to use *WinTrackInfo()* you must know more about the components of the TRACKINFO structure:

```
#define INCL_WINTRACKRECT
typedef struct _TRACKINFO
{
    // ti
    SHORT cxBorder ;
    SHORT cyBorder ;
    SHORT cxGrid ;
    SHORT cyGrid ;
    SHORT cxKeyboard ;
    SHORT cyKeyboard ;
    RECTL rcITrack ;
}
```

```

    RECTL rc1Boundary ;
    POINTL pt1MinTrackSize ;
    POINTL pt1MaxTrackSize ;
    USHORT fs;
} TRACKINFO ;

```

The members `cxBorder` and `cyBorder` let you define the width to assign to the border of the rectangle to be displayed on the screen. These quantities are expressed in pixels. With `cxGrid` and `cyGrid`, you define the minimum values that the function will consider as horizontal and vertical movements. If the movement of an object is controlled through the cursor keys, then the two members `cxKeyboard` and `cyKeyboard` indicate the number of pixels traveled in each direction. *WinTrackRect()* must also know the overall size of the object to be dragged, and the surface that will be covered by displaying it. The rectangle `rc1Track` defines the original object to be moved (in the sample, the on-screen size of a bitmap). After using this function, this member will contain the target position of the object. The second `RECTL` structure, `rc1Boundary`, defines the perimeter boundaries that delimit all possible movements. This quantity most often coincides with the size of a window on the screen or with the entire desktop window.

The minimal amount of movement that an object can make in the two directions is expressed in the `POINTL` member `pt1MinTrackSize`, while the second `POINTL`, `pt1MaxTrackSize`, describes the maximum size. The `TRACKINFO` structure is completed by a value containing the flags described in Table 5.13. These flags describe the kind of dragging that is to be performed.

In Listing 5.8 you can see a concrete example of moving a bitmap inside the client area of a window.

The program will display a bitmap, previously created with `ICONEDIT.EXE`, in a preset position of the client window. Pressing the right mouse button—and thus intercepting the `WM_BUTTON2DOWN` message— will first verify that the mouse's position is inside the rectangle occupied by the bitmap. If this test is passed, then `TRACKINFO` structure is compiled, and the `TF_MOVE` flag is set.

The starting rectangle corresponds to the bitmap's position in the client window, while the rectangle describing the dragging boundaries is equivalent to the whole client window.

When the *WinTrackRect()* is called, no presentation space handle is given, since it will be the same *WinTrackRect()* function that will take care of identifying a presentation space according to the window handle. *WinTrackRect()* does not return any value until the mouse button is released (and in the window procedure no message will be received regarding this operation). To place the bitmap in its new position it is sufficient to base the operation on the contents of `ti.rc1Track`.

In Figure 5.16 you can see how the application looks immediately after the mouse button has been pressed on some pixel of the bitmap in the client window.

As long as you keep the left mouse button pressed, you can move the bitmap outline on the screen. Figure 5.17 shows this phase of the application's execution.

**Table 5.13 Options to Instruct How *WinTrackRect()* Should Behave When Dragging an Object**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
TF_LEFT	0x0001	Drag the left border of the rectangle.
TF_TOP	0x0002	Drag the upper border of the rectangle.
TF_RIGHT	0x0004	Drag the right border of the rectangle.
TF_BOTTOM	0x0008	Drag the lower border of the rectangle.
TF_MOVE	0x000F	Drag any border of the rectangle.
TF_SETPOINTERPOS	0x0010	Repositions the pointer according to the presence of the previous flags: TF_LEFT, TF_TOP, TF_RIGHT, TF_BOTTOM, and TF_MOVE.
TF_GRID	0x0020	Limits the action of dragging to the area specified by the members cxGrid and cyGrid in the TRACKINFO structure.
TF_STANDARD	0x0040	The height and width of the grid are all multiples of the height and width of the border.
TF_ALLINBOUNDARY	0x0080	Dragging the rectangle is not allowed outside of the area identified by the rclBoundary member of the TRACKINFO structure.
TF_VALIDATETRACKRECT	0x0100	Validates the dragged rectangle.
TF_PARTINBOUNDARY	0x0200	Dragging the rectangle is partially inside the area identified by the rclBoundary member of the TRACKINFO structure.

The *WinTrackRect()* function can track all mouse movements thanks to its own message loop that operates independently of the application. This will be true until the user releases the mouse button, or presses the ESC or the Enter key. However, you can also track a rectangle without having to press the left mouse button, provided, though, that the action was initiated by some command different from pressing the left mouse button. This is what happens when you move or resize a window by selecting the Move or Size command from the title bar menu. This will move the whole window by calling the *WinTrackRect()* function. The example in Listing 5.8 lets you drag the bitmap with the keyboard cursor keys after pressing the Enter key.



## ASSOCTABLE

The ASSOCTABLE resource is a kind of compromise between a text resource and a binary resource. The outcome is evident and closely related with the operational philosophy of WPS. It allows an application to define one or more object generators

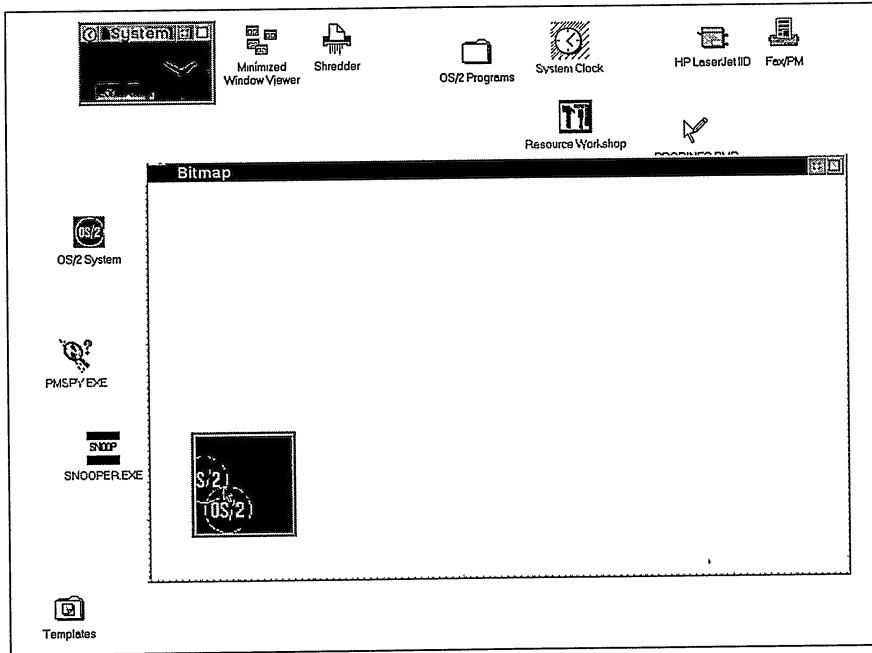


Figure 5.16 The selection of a bitmap with the mouse will cause the outline rectangle to appear around the image.

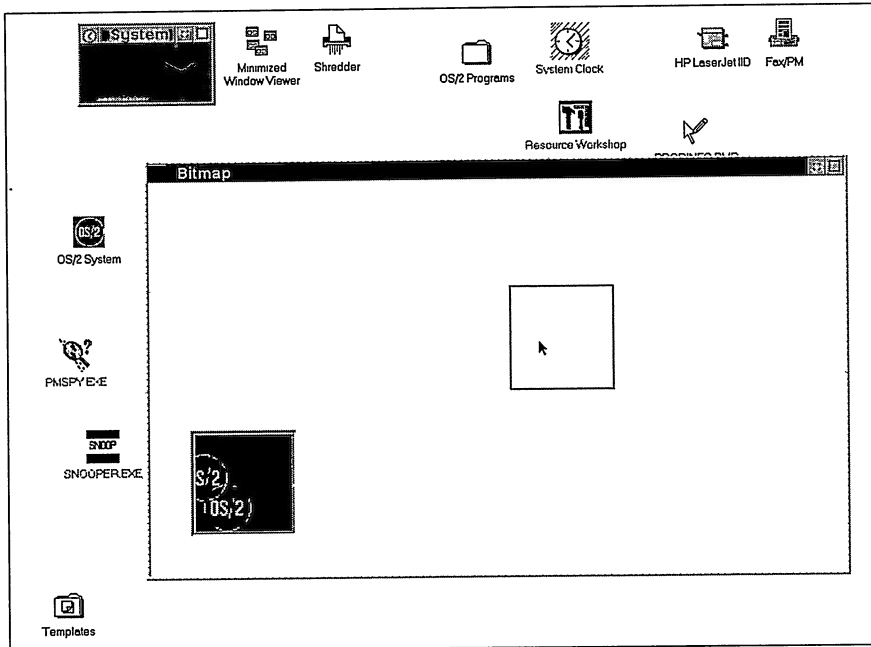


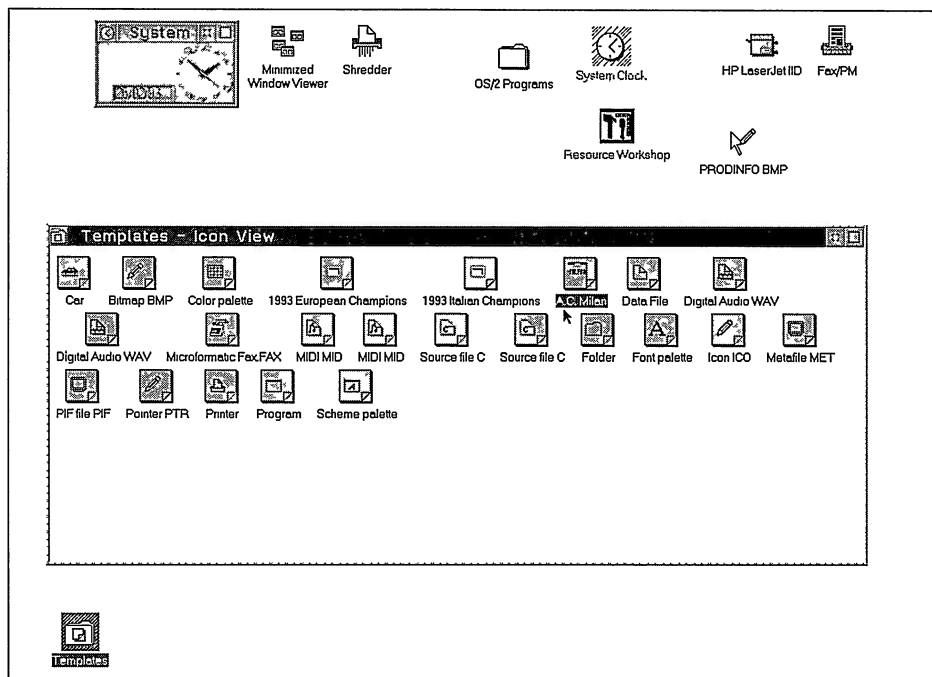
Figure 5.17 Tracking the mouse after the bitmap has been selected and after calling *WinTrackRect()*.

inside the *Templates* folder; this will allow you to originate files simply by dragging an object. Applications like Lotus 1-2-3, Describe, and many others, furnish their RC files with an ASSOCTABLE resource to extend the number of objects that can be present in *Templates*. The creation of a new Lotus 1-2-3 worksheet simply requires opening the folder and dragging the appropriate object (Figure 5.18).

The syntax of ASSOCTABLE is textual, although it will reference the name of a file containing an icon, like this:

```
ASSOCTABLE assoctableID
{
    assocname, extensions, flags, iconfile
}
```

The resource file can contain a number of association tables, and each one must be identified by a unique ID (assoctableID) defined in the header file. The body of an ASSOCTABLE must contain one or more definitions for establishing the kind of document (file) that is to be associated with an application. The syntax will support the presence of a text string enclosed by double quotes (assocname) that describes the kind of file generated by the application. Any text string is valid. However, it is a good practice to avoid overly long names because they are awkward to handle in WPS.



**Figure 5.18** The Templates folder contains a standard object and objects created by installed applications.



Table 5.14 Flags of ASSOCTABLE

<i>Flag</i>	<i>Value</i>	<i>Description</i>
EAFF_DEFAULTOWNER	0x0001	Defines the owner application.
EAFF_UNCHANGEABLE	0x0002	The definition is unchangeable.
EAFF_REUSEICON	0x0004	Allows to reuse the icon referenced in a previous ASSOCTABLE directive.

The extensions item refers to a text string (also enclosed by double quotes) that corresponds to the extension of the document files produced by the application. The classification of files according to the *name.extension* system is not a typical feature of the HPFS file system, but it is so ingrained in PC users that it will probably never disappear. If an application generates documents that are characterized by a certain constant extension (DOC, TXT, XXX, or whatever), then it is convenient to indicate it in the syntax of ASSOCTABLE. The files with that particular extension will automatically be associated with the application, and if they are displayed in WPS, they will disguise themselves with the indicated icon. The flags of ASSOCTABLE are listed in Table 5.14.

The name of a file containing the icon to associate with the specified documents completes the syntax of ASSOCTABLE. Figure 5.19 shows the *Templates* folder with the object produced by the ASSOCT application listed in Listing 5.9.

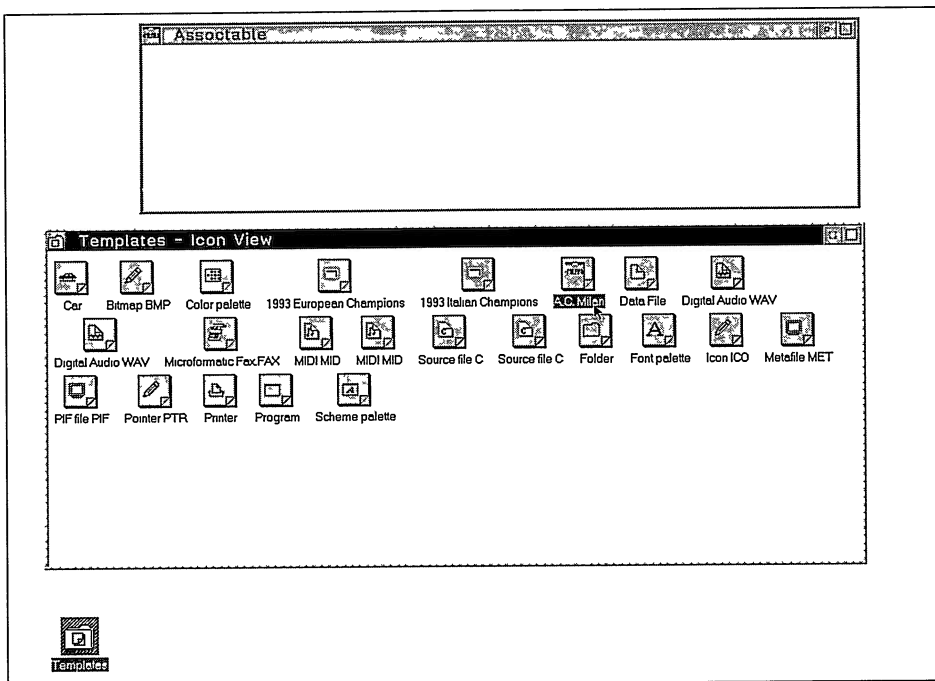


Figure 5.19 The ASSOCT application and the object produced automatically in the Templates folder.



# Menus

The term *menu* does not identify a specific item, but a series of windows that can be quite different one from another. The menu bar appears beneath the titlebar. Within this bar there are some text strings known as *top-level* menus. Generally, by selecting a top-level menu you will cause a *drop-down* menu to appear, which can, in turn, contain yet another, second-level, drop-down menu. The window to the left of the titlebar is always a menu. With the introduction of WPS, a new kind of menu has been introduced, known as the *window context menu*. The similarities to a drop-down menu are several; the only difference is that it can be positioned virtually anywhere on the screen and that its position is not related to the menu bar.

The shape, appearance, and display mode can vary substantially. The common element is that all menus are always windows that belong to the predefined class `WC_MENU`. Despite the fact that menus are actually windows, it is quite rare to use `WinCreateWindow()`, and usually only in very special cases that will be examined in this chapter and in Chapter 9. Almost invariably, the designer will prefer to define the overall structure of the menu bar and the related drop-down menus directly in the resource file, through a `MENU` directive. Yet another element that can cause some confusion has been introduced by WPS. The new interface of OS/2 2.x systems requires that the software designer modify the standard behavior of the API to follow the development model of WPS. We will discuss these situations, starting with the exploration of the titlebar menu, then proceeding to the menu bar, the menu templates, the accelerator tables, and the window context menu.

Let's take a look at Figure 6.1, containing Lotus 1-2-3 for PM. File and Edit are two top-level menus because they appear in the menu bar. To the left of the titlebar you can also see the icon of the titlebar menu.

When a top-level menu is selected, most often a drop-down menu will be displayed on the screen. Figure 6.2 shows the drop-down menu that is associated with the File top-level menu in Lotus 1-2-3.

A drop-down menu is composed of stacked options directly selectable by the user. Every single option is also known as a *menu item*. The resource file syntax provides a `MENUITEM` directive for describing these items. The selection of a top-level menu will usually display a drop-down menu from which a single option can be selected with the mouse, keyboard, or accelerator (*accelerators* are key combinations expressly defined by the software designer to make it easier for the user to perform a selection).

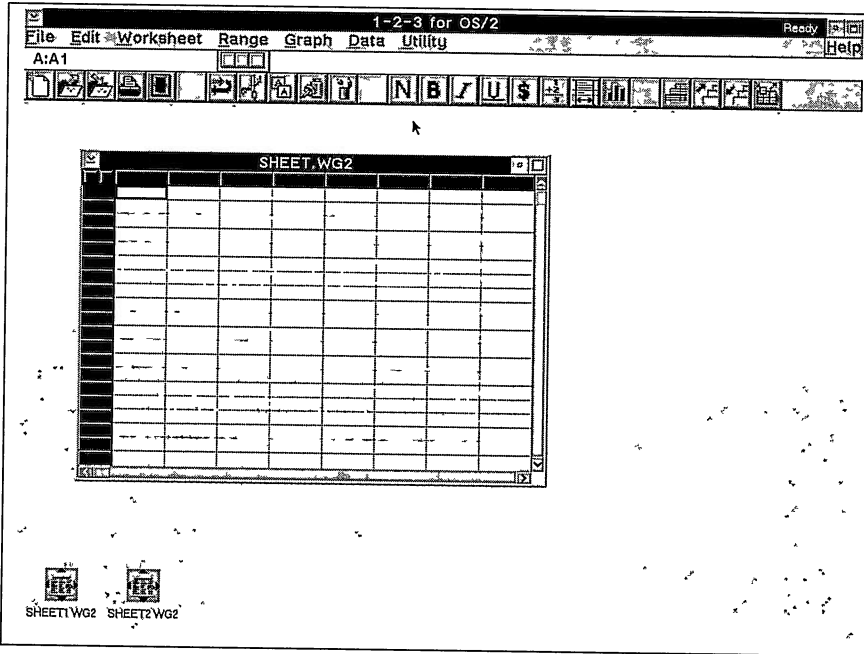


Figure 6.1 Lotus 1-2-3 2.0 for OS/2 as a sophisticated menu bar.

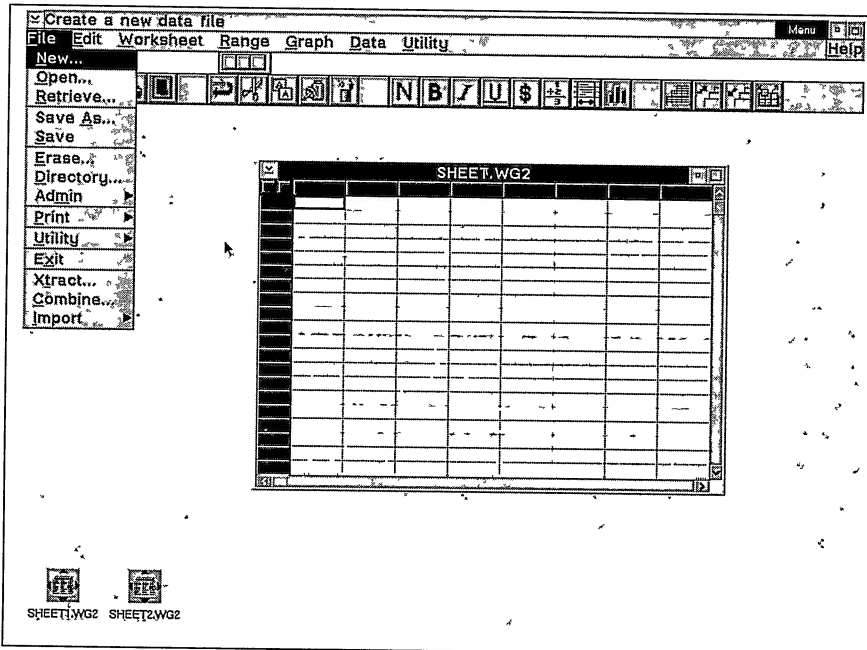


Figure 6.2 A typical drop-down menu in Lotus 1-2-3.

Each standard PM window will also have a menu to the left of the titlebar. In the 16-bit world, this was called the system menu, although the newer terminology is *titlebar menu*. This menu is characterized by a custom icon for the application and an associated drop-down menu. By pressing the left or right mouse button on this icon, or by pressing the ALT+Spacebar key combination, you will make the drop-down menu appear (Figure 6.3).

The titlebar menu will also appear whenever you press the left mouse button once over an iconized window (Figure 6.4). This situation, though, is quite rare since the introduction of the WPS interface.

The minimum common denominator among all these objects is that they all belong to the WC\_MENU class. The fact that they are windows is almost irrelevant when creating menus. PM's API furnishes the software developer with a rich set of tools for creating and managing menus, which can be used to diversify and customize applications. We will discover the true nature of PM's windows by examining a number of listings of growing complexity. What is important to notice is the differences that are often found between the API conventions and the development model adopted by WPS.

---

## The Titlebar Menu

The titlebar menu appears in every PM top-level window that has the FCF\_SYSMENU flag set (the name of this flag has not been changed so as to support backward

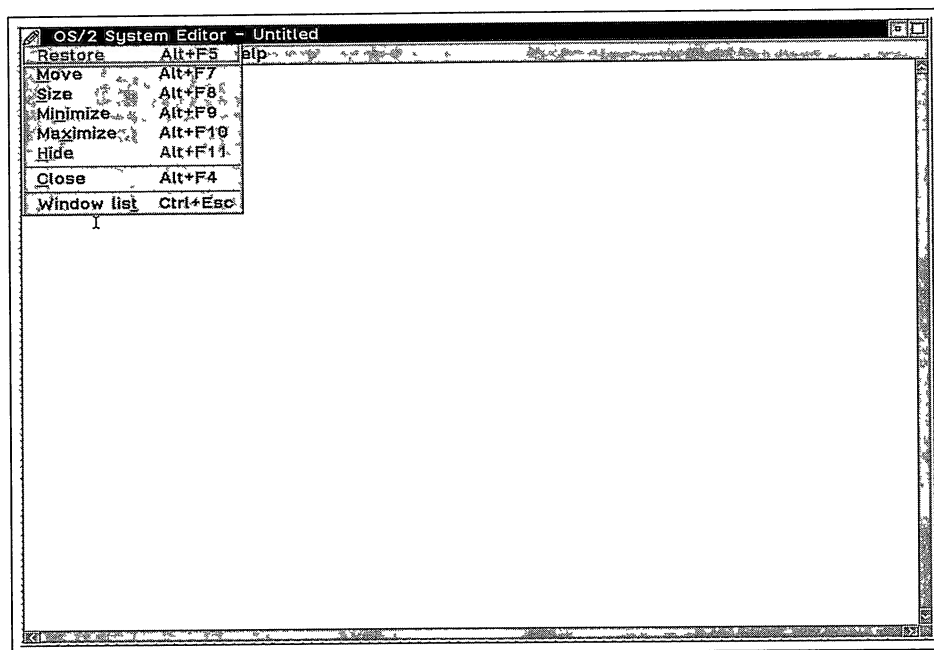
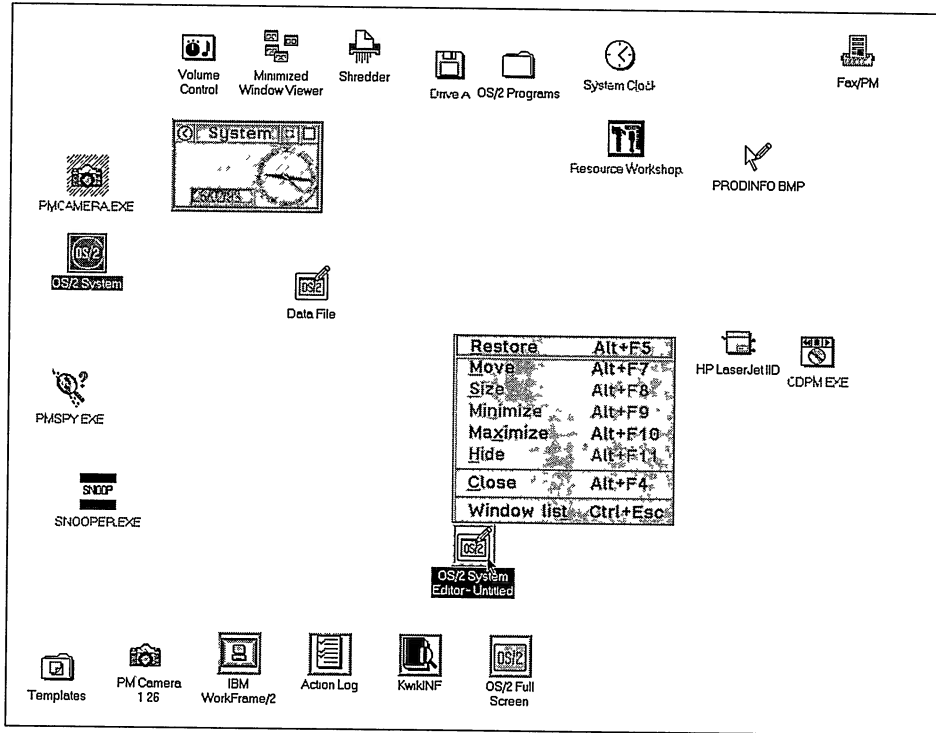


Figure 6.3 The titlebar menu of the OS/2 2.1 System Editor.



**Figure 6.4** The titlebar menu allows you to control an application also when it is minimized.

compatibility). According to style rules, each top-level PM window should provide this kind of menu to allow users to control the application even if they do not have a mouse. This can be done by setting the FCF\_SYSMENU flag as a drop-down menu containing eight options, divided into three areas by horizontal separation bars.

Often it will not be possible to select all options at all times, since the first five control the position and size of a window on the screen, and are in certain situations mutually exclusive. This is the first piece of evidence of the dynamic control of the behavior of a window according to the context in which it is used. Table 6.1 lists the menu options that are generated by setting the FCF\_SYSMENU flag.

As we have seen in Chapter 4, each structural element of a generic PM window is assigned an ID by the frame window. For the titlebar menu, this ID is FID\_SYSMENU. This means that the icon of this menu is a frame control of the WC\_MENU class. Then, there is the associated drop-down menu. Naturally, this is also a window belonging to the WC\_MENU class. Since its contents are standard (Restore, Move, and so on), each option is also assigned a unique ID, as is summarized in Table 6.2.

The selection of any of the titlebar menu options is handled by PM through the WM\_SYSCOMMAND message, addressing it exclusively to the frame window.

**Table 6.1 The Options of the Titlebar Menu in a PM Application**

<i>Options</i>	<i>Description</i>
Restore	This option is active only when a window has been maximized or minimized: its purpose is that of restoring the window to its size and position before it was maximized or minimized.
Move	The movement of a window on the screen can be performed with the mouse by keeping its left button pressed once the pointer is positioned over the window's titlebar. With Move you can achieve the same result through the keyboard. After selecting this option, the mouse pointer changes into a four-headed arrow; by using the cursor keys and then pressing Enter you can set the window's new position permanently. Move can be selected only if the window has a titlebar, that is, if the flag FCF_TITLEBAR is set.
Size	With this option you can resize a window in any of the four directions by using the cursor key. With a mouse the same action is performed directly on the screen by acting on the window's sizing border. A window can be resized only if it has the frame control flag FCF_SIZEBORDER set.
Minimize	Selecting this option is equivalent to pressing the left mouse button over the minimize icon that appears to the right of the titlebar. Minimize can be selected only if the window has been created with the flag FCF_MINBUTTON or FCF_MINMAX set.
Maximize	Selecting this option is equivalent to pressing the left mouse button over the maximize icon that appears to the right of the titlebar. Maximize can be selected only if the window has been created with the flag FCF_MAXBUTTON or FCF_MINMAX set.
Hide	This option will hide the window from the user's view, parking it in the Minimized Window Viewer container.
Close	To close a window, the only direct tool provided by the system is the selection of the Close option in the titlebar menu. By default, a double-click with the left mouse button over the titlebar menu icon will select the Close option and thus terminate the application.
Window list	The selection of this option allows you to directly access the Window List, which records all active tasks in the system.

		<i>Description</i>
WM_SYSCOMMAND	0x0021	
mp1	USHORT uscmd	One of the defines with the SC_ prefix
mp2	USHORT ussource	Source of the message.
	USHORT uspointer	Operation performed with the mouse (TRUE) or with the keyboard (FALSE)
Return Value	Reserved	

**Table 6.2 The IDs of the Options in the Titlebar Menu of PM**

<i>Option</i>	<i>Value</i>	<i>Description</i>
SC_SIZE	0x8000	Size option
SC_MOVE	0x8001	Move option
SC_MINIMIZE	0x8002	Minimize option
SC_MAXIMIZE	0x8003	Maximize option
SC_CLOSE	0x8004	Close option
SC_RESTORE	0x8008	Restore option
SC_TASKMANAGER	0x8011	Window List option
SC_HIDE	0x802a	Hide option

This message is unknown to many programmers, unless they are adventurous enough to try subclassing a frame. WM\_SYSCOMMAND reaches only a frame window procedure, never a client one. By inspecting PMWIN.H you will discover many more defines with the SC\_ prefix (Table 6.3) that complement those listed in Table 6.2. These are other IDs that can be added to the standard ones in the titlebar menu.

The number of values for the titlebar menu is an indicator of the variability of this drop-down menu's look. To see this more clearly, just display the titlebar menu of any folder of WPS, and compare it with the equivalent drop-down menu of the system editor. The difference is enormous.

## *The Titlebar Menu and WPS*

In Chapter 2 we discussed the development model to follow when creating a PM application. Menus are affected by the model, and even more particularly by the ergonomic design and behavioral rules of the program. In theory, you can choose among:

- Model CUA 89
- Model CUA 91
- WPS interface.

From the point of view adopted in this text, this is *No problema!* There should be no doubt about it: You have to develop WPS compliant applications. This choice, though, is somewhat muddled by PM's API conventions, which while enriched by new functions in versions 2.x, are still tied, for compatibility reasons, to the CUA 89 model introduced by the first versions of the operating system.

An example is given by the system menu. The API provides the FCF\_SYSMENU flag and the FID\_SYSMENU ID. It is important to note that when you create a standard window you actually obtain a drop-down that is not compliant with the rules of WPS. From this point on, we will give new meaning to the expression *generic PM application*. We will mean a program that identifies itself with a window on the screen, associated



**Table 6.3** The Defines Introduced by the `SC_` Prefix, and Used by the `WM_SYSCOMMAND` Message in `mp1`

<i>Option</i>	<i>Value</i>
<code>SC_NEXT</code>	<code>0x8005</code>
<code>SC_APPMENU</code>	<code>0x8006</code>
<code>SC_SYSMENU</code>	<code>0x8007</code>
<code>SC_NEXTFRAME</code>	<code>0x8009</code>
<code>SC_NEXTWINDOW</code>	<code>0x8010</code>
<code>SC_HELPKEYS</code>	<code>0x8012</code>
<code>SC_HELPINDEX</code>	<code>0x8013</code>
<code>SC_HELPEXTENDED</code>	<code>0x8014</code>
<code>SC_SWITCHPANELIDS</code>	<code>0x8015</code>
<code>SC_DBE_FIRST</code>	<code>0x8018</code>
<code>SC_DBE_LAST</code>	<code>0x801F</code>
<code>SC_BEGINDRAG</code>	<code>0x8020</code>
<code>SC_ENDDRAG</code>	<code>0x8021</code>
<code>SC_SELECT</code>	<code>0x8022</code>
<code>SC_OPEN</code>	<code>0x8023</code>
<code>SC_CONTEXTMENU</code>	<code>0x8024</code>
<code>SC_CONTEXTHELP</code>	<code>0x8025</code>
<code>SC_TEXTEDIT</code>	<code>0x8026</code>
<code>SC_BEGINSELECT</code>	<code>0x8027</code>
<code>SC_ENDSELECT</code>	<code>0x8028</code>
<code>SC_WINDOW</code>	<code>0x8029</code>

with a titlebar menu rather than a system menu. Then there is the *window context menu*, a menu that is identical to the titlebar drop-down, though it is displayed anywhere in the client. Often, the titlebar menu can enhance the look of a window. Furthermore, and it is important to be aware of this, the chosen development model is that of SDI.

It is very convenient to get a menu displayed by pressing the right mouse button any place over an object. WPS has accustomed users to operate this way. This operation, however, does not produce any effect in the system editor and in others items in the Productivity folder. The system editor and almost all of the *applets* provided with the system are old 16-bit applications that were developed before the release of OS/2 2.x, or that are not compliant with the new style rules. As such, they should not be followed.

There is a fundamental inconsistency between what you wish to do with menus, and the effects produced by the API. Let's examine the APIs regarding the interaction with the system menu; then we will analyze what is necessary to change it into a

titlebar menu, that is, into an object compliant with the rules of WPS. There is a lot of work to do and many things to discover here. All new applications written for OS/2 2.1 must follow the rules of the WPS interface, which is, effectively, a new way of using a PC. Menus play a crucial role.

## The APIs of PM and the System Menu

The icon that appears to the left of the titlebar is actually a window belonging to the WC\_MENU class, and is identified by the FID\_SYSMENU ID. This window provides an area for the user to interact with the application, and for the application to display the drop-down menu that is known, according to the terminology introduced by CUA 91, as the titlebar menu. This menu can present a wealth of options (Figure 6.5).

By pressing the right or left mouse button, you will display the drop-down menu, which is also a window of the WC\_MENU class. The concept that you must understand is this: We are dealing with windows. For the first time, though, these are not windows belonging to classes registered by the code, but are representatives of one of the fifteen predefined PM classes. You need to adopt a different strategy with respect to what you have already learned, because you cannot control the class's window procedure directly.

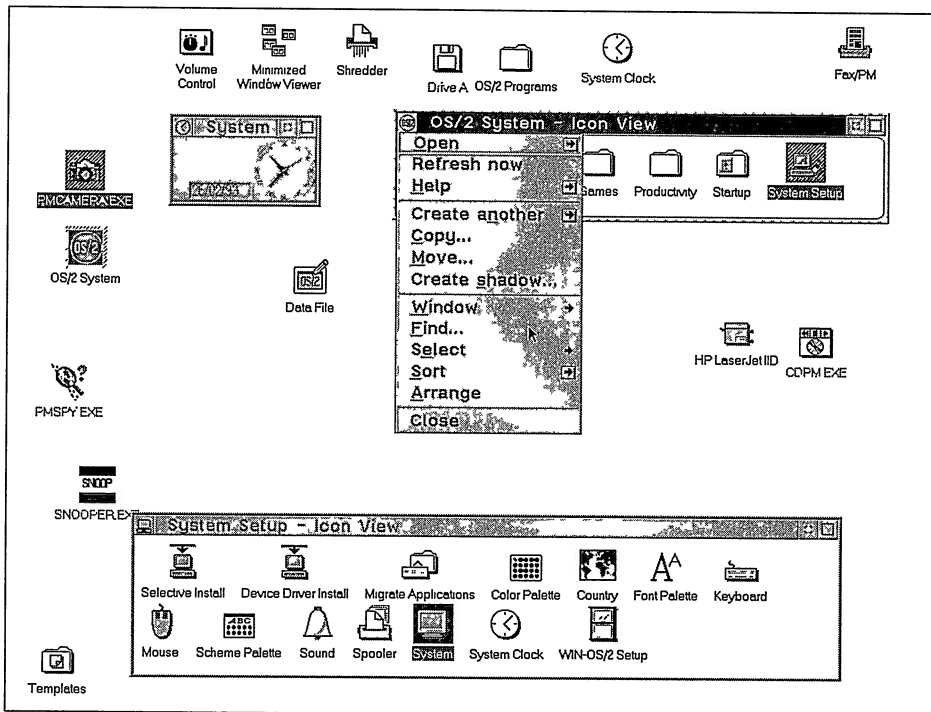


Figure 6.5 The titlebar menu in the OS/2 System folder.

Using the `FCF_SYSMENU` flag frees the designer from all problems. To the left of the titlebar, the standard icon will be displayed. If this icon is selected, it will present the drop-down menu containing the options described in Table 6.1. We can call this menu the *system menu*. As such, it is good programming practice never to change it, leaving in it the ordinary options for handling the window's positioning and sizing. It is unlikely that a user with a mouse will ever lose time by selecting any of these menu items. Most often, the only interaction with the system menu will be to double-click on its icon, an operation which is equivalent to selecting the Close option. This will close the window and terminate the application.

The status of the options in the system menu is handled automatically by PM. When a window is maximized, the Maximize options will no longer be selectable. A simple customization is that of replacing the standard icon with another one. This is what happens, for instance, in the system editor. The function `WinCreateStdWindow()` allows you to retrieve the application's own icon from a resource file, and this icon replaces the standard icon for the system menu. If you just want a simple system menu, the only code you need to know is that for setting a `FCF_SYSMENU`.

---

## Giving a Menu to a Window

Let's have a look at the menu bar. A menu bar (or *action bar*) is simply a window. This window can be created by calling the `WinCreateWindow()` function, as in all examples seen in the preceding chapters. It is also a resource according to the development model of PM. It is, however, preferable to delegate the generation of a menu to PM's API by reading and loading a *menu template* defined in the resource file. The rules to follow when writing a menu template are very simple, and allow for the easy maintenance and updating of an application's menu structures.

The first operation when creating a menu is defining a menu template. Once the menu template has been created, load it and transform it into a window by taking advantage of PM's API. Even in this case, there are a number of solutions available to the application's objectives. Once a window has been provided with a menu, it is necessary to modify the source code in order to catch the messages generated by the system when the user makes selections.

## Creating a Menu Template

The IBM Toolkit does not provide any tool for creating menu templates in a simple way. Therefore, you must type the structure of a menu template directly in a resource file or in some other ASCII file. As we have seen in Chapter 5, a resource in a RC file is almost always characterized by an ID; this is also true for the `MENU` directive for the resource compiler.

The syntax of the `MENU` directive requires, in addition to the ID (chosen in the range between 0 and 64KB), that you also specify loading options and memory management options.

```

MENU menu-id [load options][memory management options]
{
    ...
}

```

The ID is essential for its subsequent use with *WinCreateStdWindow()* or with *WinLoadMenu()*. By using *WinCreateStdWindow()*, the generation of a menu bar is completely transparent to the programmer. The function itself performs all the operations of reading the template and transforming it into a series of windows (the menu bar and the associated drop-down menus). In Chapter 4 we examined the FCF\_ flags involved in the creation of frame controls like the titlebar, the sizing border, the system menu, and others. In the examples thus far, we have used the FCF\_STANDARD flag for the *WinCreateStdWindow()* function, and subtracted from it the values corresponding to FCF\_MENU, FCF\_ICON, and FCF\_ACCELTABLE, according to the various requirements.

These three frame control flags govern the generation of those elements that can adorn a window but that are not necessary for the application's functioning. The second common feature is their operation. Each references a resource declared in the RC file. To create a window equipped with a MENU, ICON, and ACCELTABLE resource, you must assign the same ID in order to ensure that *WinCreatStdWindow()* works correctly when the window is created. In fact, the presence of any one of these three flags, and the absence of its corresponding resource in the RC file will cause a run-time error for the application and prevent it from appearing on the screen. In order to avoid this, the creation of a menu template and its subsequent use with the FCF\_MENU flag in *WinCreateStdWindow()* will automatically generate a menu bar in a PM window.

With *WinLoadMenu()* you can instead load a menu template from an application's resource file. The value returned by this function is a handle to the menu, and it is this handle that allows you to manage the object and use it effectively in the program.

```

#define INCL_WINMENUS
HWND APIENTRY WinLoadMenu(  HWND hwndFrame,
                             HMODULE hmod,
                             ULONG idMenu) ;

```

<i>Parameter</i>	<i>Description</i>
hwndFrame	Handle to the frame window with which the menu is associated
hmod	Handle of the module from which the resource is to be loaded
idMenu	ID of the menu template to load
<i>Return Value</i>	<i>Description</i>
HWND	Handle of a window belonging to the class WC_MENU or NULLHANDLE in case of error

This function loads a menu template from the resource file, creates a menu type window, and displays it on the screen immediately below the titlebar, and thus reduces the area available to the client window. To do this, though, it is necessary that the first parameter corresponds to the handle of a frame window that, at that moment, is missing a control with the FID\_MENU ID.

The ID indicated as the function's third parameter must correspond to the ID specified in the resource file with the MENU directive. The second parameter, the handle of the module, defines which executable module the resource should be loaded from. Remember, in the development model of PM, the resource file effectively becomes part of the executable file, once it has been compiled. The handle hmod must always reference an EXE file or DLL.

The value assigned as the ID of a menu template must be an integer from 0 and 64Kb. Often, though, it is convenient to resort to a definition through the #define directive to make it easier to maintain the application.

```
...
MENU RS_MENU
{
    ...
```

where RS\_MENU is defined like this in the application's header file:

```
...
#define RS_MENU RS_ALL
...
```

This kind of definition corresponds to the strategy outlined in Chapter 5 for identifying a resource of type MENU, ICON, or ACCELTABLE, that needs to be associated directly with a window at the time of its creation. The MENU resource is not limited to this particular case, as we will see, after getting to know the *window context menu*.

The load options indicate how the system loader should act when the application is loaded into memory. Table 6.4 summarizes the two possible options. The resource will be handled in memory according to what is specified for the memory management options (Table 6.5).

The default combination for a generic MENU resource is LOADONCALL, MOVEABLE, and DISCARDABLE.

## The Menu Template

In the block enclosed by the pair of braces of the MENU directive, you have to insert the SUBMENU and the MENUITEM directives. From a syntactical point of view, the action bar's

**Table 6.4 Load Options for a Resource Loaded into the System's Memory**

<i>Option</i>	<i>Description</i>
PRELOAD	The resource is loaded by the system at the time the application is executed.
LOADONCALL	The resource is loaded by the system only when the application calls <i>WinLoadMenu()</i> or some other function for creating a window.

**Table 6.5 Memory Management Options for a Resource Loaded into the System's Memory**

<i>Option</i>	<i>Description</i>
FIXED	The resource is loaded and always kept in memory at some fixed address.
MOVEABLE	The resource is positioned in memory according to the needs of PM's <i>memory manager</i> ; it can be moved if it is necessary to avoid excessive memory fragmentation problems.
DISCARDABLE	The resource can be relinquished by the system if there is a request for more memory than what is available at the moment; the <i>discarding</i> operation consists of the resource's memory image being destroyed, and thus the need of referencing the original copy present in the executable file for any subsequent reference.

top-level menus can be defined both with a SUBMENU as well as with a MENUITEM directive. In the first case, in addition to defining a new top-level menu, you also have to create the drop-down menu that must appear on the screen when the user selects the corresponding top-level menu. With MENUITEM, the selection of a top-level menu corresponds to the direct execution of an action without any other intervening drop-down menu.

The CUA 91 specs strongly advise that you define top-level menus with a SUBMENU, and defer the usage of MENUITEM only for describing the *drop-down* menus.

```
MENU menu-id [load option][memory management options]
{
  SUBMENU top-level name, ID
  {
    ...
  }
  ...
}
```

The name you give to the SUBMENU directive—`top-level name`—is a text string enclosed by double quotes, and it is the name that will subsequently be displayed in the menu bar. The ID must be a value in the range 0–64Kb. When selecting an ID, it is necessary to use unique numbers within each block. It is acceptable, as far as the resource compiler is concerned, that two menu items belonging to two different drop-down menus share a same ID. What is essential is that the uniqueness is imposed within each drop-down and among all top-level menus. The same IDs can be used for different resources in an RC file. Sometimes, though, it is better to assign the same ID to internal items of two distinct resources. This is the case, for instance, of a string

specified in a STRINGTABLE and a MENUITEM. This solution is permitted and even advisable when the application window presents an information area.

The description of a drop-down menu takes place when you declare several MENUITEM directives inside the block enclosed by braces after a SUBMENU directive.

```
MENU menu-id [load options][memory management options]
{
  SUBMENU top-level name, ID
  {
    MENUITEM option name, option ID [, [style], [attribute]
    ...
  }
  ...
}
```

The option name is a text string within double quotes that corresponds to the text to be displayed in a drop-down menu, while the ID is always an integer between 0 and 64Kb. Tables 6.6 and 6.7 list and describe all the styles that you can use to customize the look of a drop-down menu.

In most cases, menuitems are simply text strings without the MIS\_TEXT style, which is implied. Most of the styles listed in Tables 6.6 and 6.7 should not be used because they are in contrast with the menu design style rules. The only styles used regularly are MIS\_SYSCOMMAND, MIS\_BITMAP, MIS\_OWNERDRAW, and MIS\_HELP. For others, like MIS\_SEPARATOR, it is much more convenient to resort to a SEPARATOR directive to produce a horizontal separator bar.

The menu attributes are limited in number (Table 6.7) and directly affect the look of each single MENUITEM present in a SUBMENU block.

Tables 6.6 and 6.7 stress the fact that all styles and attributes refer to options introduced by a MENUITEM directive in a SUBMENU block. Previously we did not discern any greater difference between a MENUITEM within a SUBMENU, in contrast with it appearing immediately below a MENU directive. In fact, MENUITEM can originate a top-level menu free of any drop-down menu. In this case, though, the text string corresponding to the menu (or, more rarely, the bitmap), cannot present any further style modification.

The attributes that are used most frequently are MIA\_DISABLED to disable a menu item and MIA\_CHECKED to draw a check-mark symbol to the left of the item's text.

## *Keyboard Access*

An OS/2 system without a mouse is almost an insult to common sense. However, the CUA 91 specifications cater to those users who do not have a mouse or who are still uncomfortable with the friendly desktop rodent.

Each option in a menu window—including all those for all kinds of menus examined up to this point—must allow for keyboard combinations that let the user make a selection. The titlebar menu is activated by the sequence ALT+Spacebar. The first top-level menu (the leftmost one in the menu bar) is accessed by pressing the ALT key

**Table 6.6 Styles for Customizing and Modifying the Aspect of a Drop-Down Menu**

<i>Style</i>	<i>Value</i>	<i>Description</i>
MIS_TEXT	0x0001	Default style for any MENUITEM; it is not necessary to set it explicitly.
MIS_BITMAP	0x0002	Alternative style for a drop-down menu option: Instead of describing a MENUITEM with a text string, you can use a bitmap.
MIS_SEPARATOR	0x0004	Predefined style that will make a horizontal separator bar appear in a drop-down menu. In place of using this style flag, which will always require you to specify a text string to qualify the menu option, you might prefer to use a MENUITEM SEPARATOR directive, which produces the same result.
MIS_OWNERDRAW	0x0008	With this flag you can inform the menu window that you wish to handle all drawing operations of the menu options and drop-down menus yourself. This control will be passed to the owner window of the menu through the message WM_DRAWITEM followed by a WM_MEASUREITEM.
MIS_SUBMENU	0x0010	This flag is used to indicate that a certain option takes on the role of a SUBMENU: The use of this flag is limited to the dynamic creation of a menu, rather than referencing a template in the resource file.
MIS_MULTMENU	0x0020	This flag allows you to create a menu spanning multiple lines. According to the CUA specifications, it is preferable not to use this layout when structuring a menu template.
MIS_SYSCOMMAND	0x0040	This style will modify the application's standard behavior in response to a user's selecting a menu option; the action will issue a WM_SYSCOMMAND message rather than the typical WM_COMMAND.
MIS_HELP	0x0080	This style modifies the application's default behavior in response to the user's selection: In this case the code will receive a WM_HELP message when the user selects an option that has this flag set.

*continued*



**Table 6.6 (Continued)**

<i>Style</i>	<i>Value</i>	<i>Description</i>
MIS_STATIC	0x0100	You can set this flag to avoid a drop-down menu text option being selected via the keyboard or the mouse: The text appears simply like a static piece of information and will not issue any message to the application.
MIS_BUTTONSEPARATOR	0x0200	This style was used in version 1.x exclusively for a help menu, since it will center the text inside a menu rather than left-align it.
MIS_BREAK	0x0400	This style indicates that a drop-down menu vertically spans multiple lines instead of one: The use of this flag is rare, mainly because the CUA specifications recommend that drop-down menus should appear in one column only.
MIS_BREAKSEPARATOR	0x0800	This style is used in combination with the previous one: Its action is limited to displaying a vertical bar between two neighboring columns in a multi-column drop-down menu; the same considerations expressed above still hold.

**Table 6.7 The Menu Attributes in PM**

<i>Attribute</i>	<i>Value</i>	<i>Description</i>
MIA_NODISMISS	0x0020	Prevents a drop-down menu from disappearing after the selection of an option: The parent will still remain the desktop.
MIA_FRAMED	0x1000	Encloses the option within a rectangular border.
MIA_CHECKED	0x2000	Draws a check-mark symbol to the left of the indicated MENUITEM.
MIA_DISABLED	0x4000	Disables the MENUITEM preventing the normal sending of the message WM_COMMAND, WM_SYSCOMMAND, or WM_HELP to the application: The selection via the keyboard or the mouse will be ineffectual.
MIA_HILITED	0x8000	The "highlight" in OS/2 2.1 consists of displaying the selected menuitem with a three-dimensional "sculpted" look.

or the F10 function key. Furthermore, all top-level menus and all menu items of the drop-down menus, provide a secondary form of access given by the pair of keys ALT plus a letter key chosen by the designer when writing the template.

The PM development style rules recommend that the pressing of the ALT key followed by a letter allow access to a top-level menu, and its associated drop-down menu (if any). The software designer must therefore pick a different letter for each top-level menu in order to provide the system with an unambiguous way of referencing each menu present in the menu bar. The chosen letter is known as *mnemonic code* and appears underlined on the screen. The ASCII code of 126—the tilde ~—is the escape symbol used by the resource compiler to demarcate the mnemonic code for accessing a menu via the keyboard. For instance, in the case of a File menu—which is almost invariably the first one in the menu bar—mnemonic code is the letter F: therefore ALT+F will cause the File menu's associated drop-down menu to be displayed. In the menu template the letter F is preceded by a tilde.

```
MENU RS_MENU
{
  SUBMENU "~File", MN_FILE
  {
    MENUITEM "~New", MN_NEW
    ...
  }
  ...
}
```

A drop-down menu can then have several options: In this case it is necessary to pick out a unique letter for each MENUITEM in order to allow for its selection through the keyboard. There is a fundamental difference between the mnemonics for top-level menus and those for the options in a drop-down menu. To access the first ones, you have to press the ALT key and then any of the underlined letters (like F in the File menu). On the other hand, to select an option in a drop-down menu that is already displayed on the screen, it is sufficient to press the underlined letter (like N for New).

## Menu Style Rules

In order to define the layout of a menu in a PM application, it is necessary to adhere to some style conventions and follow some practical rules. In the first place, you should never develop a number of top-level menus larger than the action bar. The menu bar must only occupy one row alone, corresponding to the window's width. The user is familiar with this kind of interface, and there is no sound reason for changing this practice.

Drop-down menus should not be too large, both in terms of number of items as well as in terms of overall width. It is not possible to determine an optimal number of options or a limit to the characters that make up an option. However, what is important is to be aware of what happens in the application's client window when a drop-down menu is displayed. (Since the environment is multitasking and multithreaded, it is

always possible that the contents of the client window may change dynamically, even if the user is engaged in the selection of a menu option.)

Furthermore, the options that appear in the various drop-down menus that characterize an application should fall in one of three categories: *commands*, *extended commands*, and *state setters* (Table 6.8).

In some applications the menu bar will have top-level menus ending with an exclamation mark. This is a convention used to indicate that the selection of that top-level item will not cause any other drop-down menu to be displayed. The exclamation mark (!) does not conform to the modern development model inspired by the CUA 91 specifications, and thus is an obsolete style rule that should not be followed. This kind of menu item can easily be transformed into an option inside a drop-down menu (maybe setting it apart from the other options by using a horizontal SEPARATOR bar).

A final consideration relates to the positioning of the help menu in the menu bar. According to the CUA 91 specifications the help menu must be placed as the last, rightmost menu in the menu bar, although often it appears to the extreme right. Yet another important aspect introduced by WPS should not be overlooked. All folders lack a menu bar! This is not an oversight, but a precise implementation choice of the designers of OS/2, which contrasts with the rules of CUA 91. We will get back to this subject at the end of this chapter.

**Table 6.8 The Three Categories That Classify the Menu Items That Can Appear in the Menu Structure of a PM Application**

<i>Category</i>	<i>Description</i>
Commands	Any option that will cause the immediate execution of an action on part of the application. The text string that defines the option must present a unique underlined pick letter, but no other special style attribute.
Extended Commands	This option will ultimately execute a command, but it will require the user to fill in a dialog in order to define all the details that affect the command to be executed. The text string of these menu items are always terminated by three periods (ellipsis) in order to indicate that a dialog window (see Chapter 8) will appear after their selection.  A typical example of this category is the Open... command: by selecting this option you will see a dialog window that allows you to select the file to be loaded, and the drive and the directory where it is located. The performed action is rich in flexibility.
State Setters	There are some options that behave like toggle switches. For example, in EPM the Stream Editing menu (Figure 6.6).

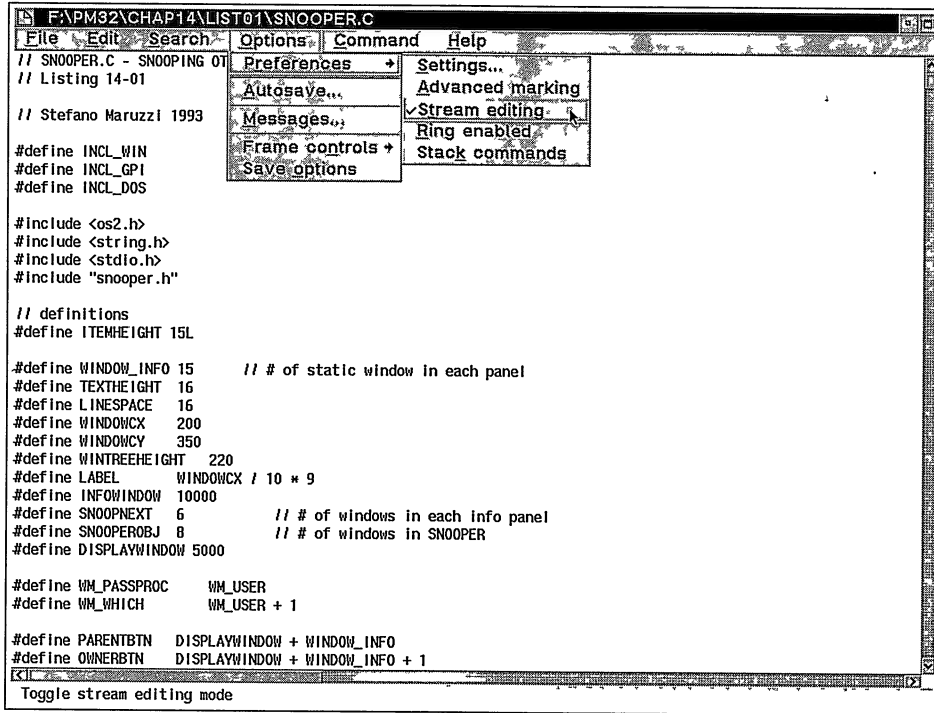


Figure 6.6 The Stream Editing menu in EPM is a typical state setter.

## Defining Templates

Now that you know all about basic tools and fundamental style rules, you can go ahead and create a menu template to be inserted directly in a resource file (Listing 6.1).

### Listing 6.1 A Menu Template in a Resource File

```
// CNTXT.RC

#define INCL_WIN
#include <os2.h>
#include "cntxt.h"

ICON RS_ICON CNTXT.ICO
BITMAP 100 CNTXT.BMP

MENU RS_MENU
{
    SUBMENU "~File" , MN_FILE
    {
        MENUITEM "~New", MN_NEW
```

```

MENUITEM "~Open", MN_OPEN
MENUITEM "~Save", MN_SAVE,, MIA_DISABLED
MENUITEM "Save ~As...", MN_SAVEAS,, MIA_DISABLED
MENUITEM SEPARATOR
MENUITEM "E~xit", MN_EXIT
}
SUBMENU "~Edit" , MN_EDIT
{
    MENUITEM "~Undo\tAlt+Backspace", MN_UNDO,, MIA_DISABLED
    MENUITEM "~Redo\tShift+Alt+backspace", MN_REDO,, MIA_DISABLED
    MENUITEM SEPARATOR
    MENUITEM "Cu~t\tShift+Delete", MN_CUT,, MIA_DISABLED
    MENUITEM "~Copy\tCtrl+Insert", MN_COPY,, MIA_DISABLED
    MENUITEM "~Paste\tShift+Insert", MN_PASTE,, MIA_DISABLED
    MENUITEM SEPARATOR
    MENUITEM "~Delete\tDelete", MN_DELETE,, MIA_DISABLED
    MENUITEM SEPARATOR
    MENUITEM "~Find...", MN_FIND,, MIA_DISABLED
    MENUITEM "~Select all\tCtrl+/", MN_SELECTALL,, MIA_DISABLED
    MENUITEM "~Deselect all\tCtrl+\\", MN_DESELECTALL,, MIA_DISABLED
}
SUBMENU "~Help", MN_HELP
{
    MENUITEM "Help ~index", MN_HELP1, MIS_HELP
    MENUITEM "~General help", MN_HELP2, MIS_HELP
    MENUITEM "~Using help", MN_HELP3, MIS_HELP
    MENUITEM "~Keys help", MN_HELP4, MIS_HELP
    MENUITEM SEPARATOR
    MENUITEM "~Product information", MN_PRODINFO
}
}

```

In this portion of a resource file you will only have two `#include` directives, in addition to the template itself: the first one refers to `OS2.H` and can even be absent because no defines present in that file are ever referenced in this resource file. However, its presence is justified by the subsequent additions that we will make to this file.

The second include file—`CNTXT.H`—is specific for this application and contains essential data both for the resource compiler and for the following execution of the C compiler. Each `SUBMENU` and `MENUITEM` directive in Listing 6.1 has an ID of its own, and that ID is defined in this specific header file (Listing 6.2).

### Listing 6.2 The Header File for the Resource File Listed in Listing 6.1

```

// CNTXT.H

#define RS_ALL          300
#define RS_ICON        RS_ALL
#define RS_MENU        RS_ALL

```

```

#define RS_ACCELTABLE    RS_ALL
#define RS_TBMENU       350

#define ST_CLASSNAME    10
#define ST_WINDOWTITLE  11

#define MN_FILE 99
// Defines for MN_FILE menu
#define MN_NEW  100
#define MN_OPEN 101
#define MN_SAVE 102
#define MN_SAVEAS 103
#define MN_EXIT 104

#define MN_EDIT 200
// Defines for MN_EDIT menu
#define MN_UNDO 201
#define MN_CUT  202
#define MN_COPY 203
#define MN_PASTE 204
#define MN_CLEAR 205

#define MN_HELP      300
#define MN_HELP1     301
#define MN_HELP2     302
#define MN_HELP3     303
#define MN_HELP4     304
#define MN_PRODINFO  305

```

The numbers used do not conform to any special rule of PM; they can be defined simply according to personal preference, like assigning sequential IDs starting from each top-level menu and including each menu item, and even the SUBMENUs in a SUBMENU. The ample choice in assigning these IDs makes it easier to define small blocks of sequential values for the single objects, like the File and the Edit menus. Furthermore, a conventional approach in writing a menu template is that of using strings with the prefix `MN_`, just to remind you that there are menu items, followed by the text that appears within the double quotes, or an abbreviation of that text. This approach makes it a lot simpler to interpret the labels even in the source code, and gives the standard look to PM applications, facilitating later maintenance of the code.

It would be possible to insert the defines contained in the header file directly in the resource file, but, as we will see later, this is not sound practice.

Each ID in the resource file listed in Listing 6.1 will have a corresponding define in the header file; the whole menu template is identified by `RS_MENU`, while the two SUBMENUs File and Edit are identified by `MN_FILE` and `MN_EDIT`. When this menu template is loaded in an application, it will generate two top-level menus, File and Edit, starting from the left side of the menu bar.

By carefully examining Listing 6.1, you will have noticed the presence of the `\t` symbol inside the text strings of the Edit menu. This notation indicates a tab character that serves to align the following portions of text on a virtual tab stop inside the drop-down menu. The creation of a menu window, in fact, follows the structural scheme outlined in Figure 6.7.

PM will align all text or bitmaps that appear in a drop-down menu in three columns. The leftmost column is left free to hold the potential check-mark symbol for state setter options. The central column, which will be of varying size depending on the text that is actually inserted, will accommodate the string or the bitmap that identifies each `MENUITEM`. The third, rightmost column will align all accelerators, and the arrow symbol for indicating that a drop-down menu option is associated with yet another `SUBMENU`. The second `SUBMENU` is displayed automatically next to the arrow that indicates it (Figure 6.8).

With the advent of WPS, a new kind of subordinate conditional menu has also been introduced. It is always a second level menu, indicated by an arrow inside a small button. The technical term is *mini-pushbutton* (Figure 6.9).

## Complex Menu Templates

The typical structure of PM applications provides for the presence of multiple top-level menus associated with drop-down menus. Each drop-down menu must correspond, at the menu template level, to a `SUBMENU` directive. A drop-down menu can in turn contain another drop-down, and so on. The structure of the menu template becomes accordingly more elaborate, compared to what you have seen so far.

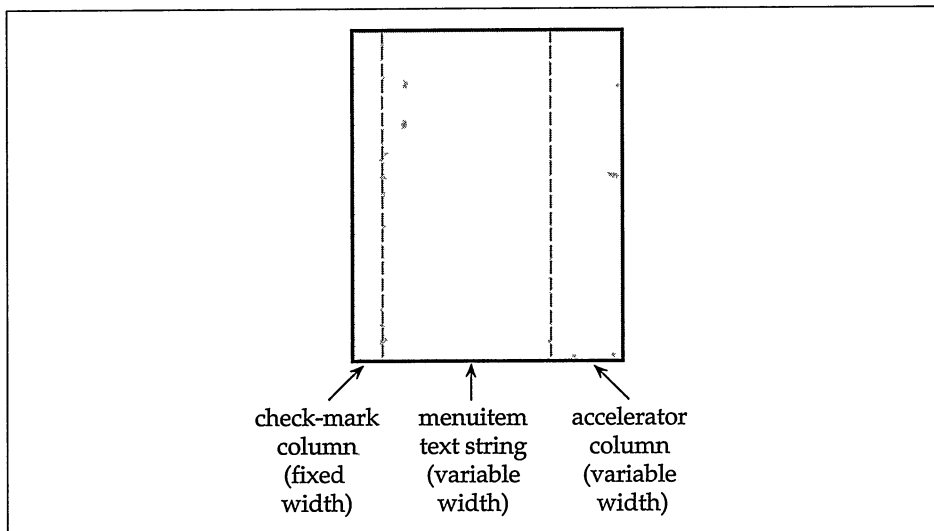
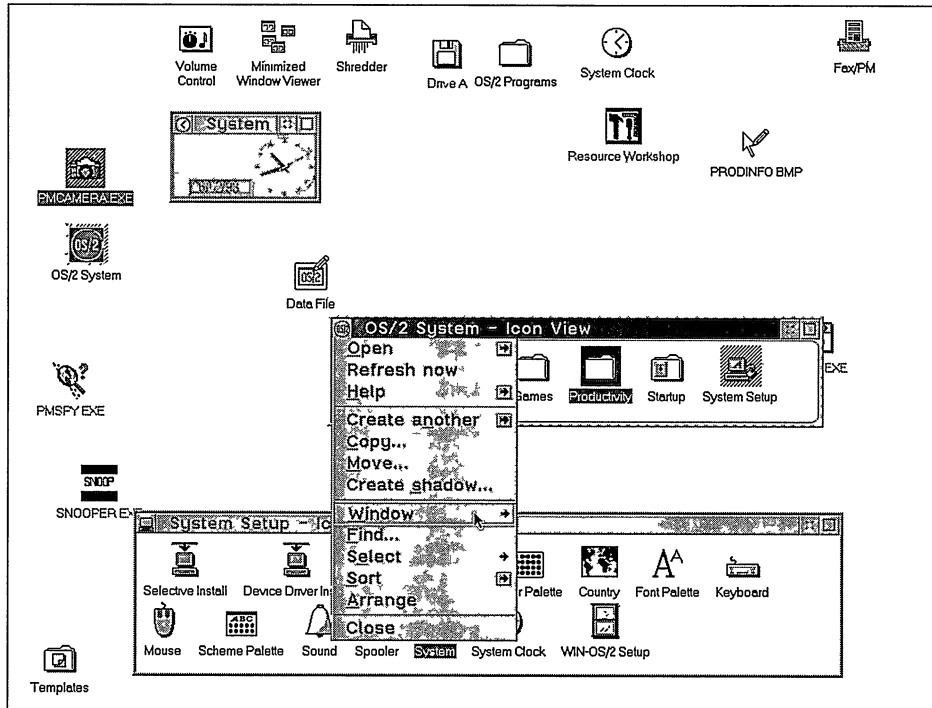


Figure 6.7 Structural scheme of a drop-down menu in PM.



**Figure 6.8** A SUBMENU of a SUBMENU is indicated by the presence of an arrow in a MENUITEM; its selection will cause the second drop-down menu to be displayed.

```

...
MENU RS_MENU
{
    ...
    SUBMENU "~Options" , MN_OPTIONS
    {
        MENUITEM "~Search" , MN_SEARCH
        MENUITEM "~Replace", MN_REPLACE
        MENUITEM "~Goto", MN_GOTO

        SUBMENU "~Fonts", MN_FONTS
        {
            MENUITEM "~Helvetica", MN_HELVETICA
            MENUITEM "~Courier", MN_COURIER
            MENUITEM "~Times", MN_TIMES
        }

        SUBMENU "~Background", MN_BACKGROUND
        {
            MENUITEM "~White", MN_WHITE
            MENUITEM "Re~d", MN_RED
            MENUITEM "~Blue", MN_BLUE
        }
    }
}

```



```

    }
    ...
}
...
}
...

```

The Fonts string appears in the drop-down menu that the application displays when Options is selected. To the far right of the MENUITEM, a right-pointing arrow means that there will be yet another drop-down menu if the user selects Fonts (Figure 6.10).

The complexity is limited only to the presence of multiple SUBMENUs inside the menu template, while no extra effort is required on part of the programmer. The previous resource file fragment does not provide for any style (MIS\_) or attribute (MIA\_) for the purpose of making the tree structure of the various blocks stand out better. However, it is good programming practice to customize a template in order to give it exactly the look it should have for the application. The presence of the Search option suggests that this part of the template refers to a program used for editing text. It can be assumed that when the program is loaded there will be no text available to edit, and thus the Search option could initially be disabled, and then enabled as soon as a document is loaded or some text is typed in.

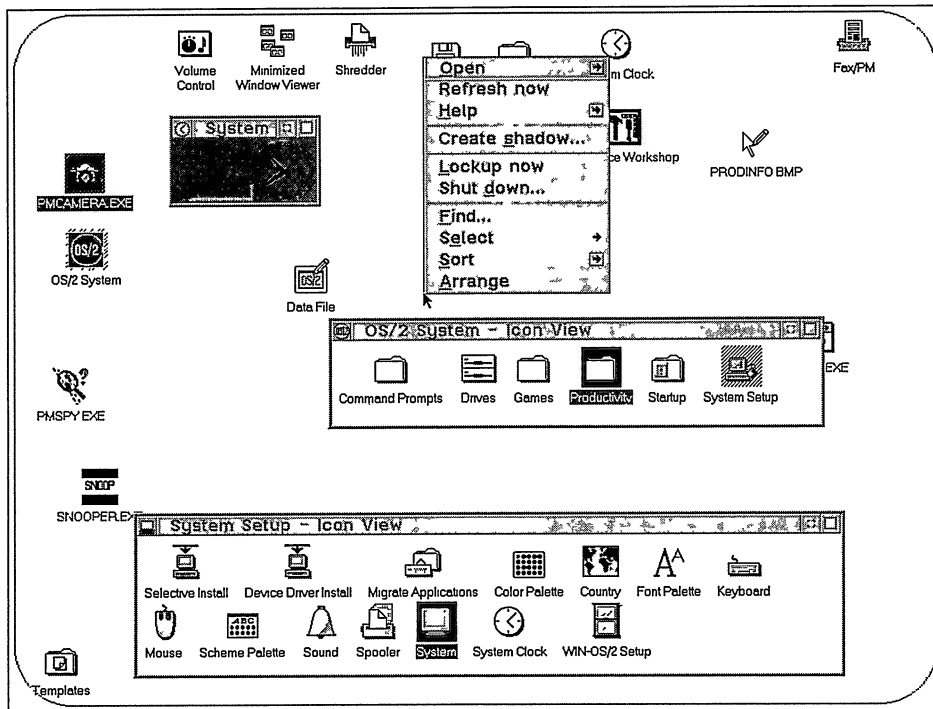


Figure 6.9 The window context menu of the desktop of an OS/2 2.1 system will have three menu items equipped with subordinate conditional menus.

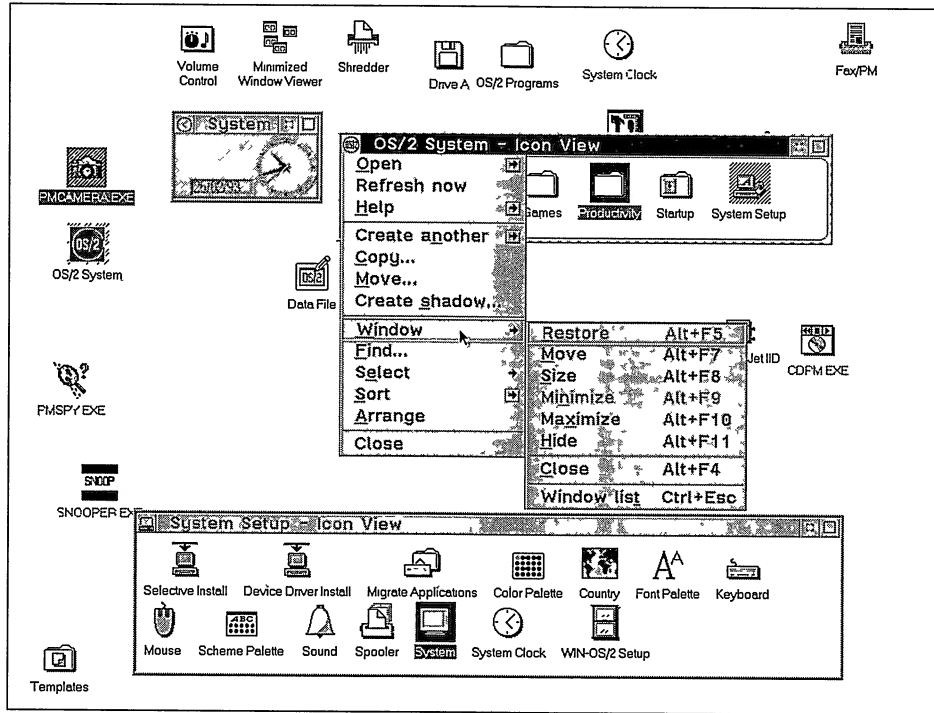


Figure 6.10 The structure of a drop-down menu that provides for a second drop-down menu will display a small horizontal arrow.

## Syntax Rules for Menu Templates

Considering the crucial role played by menus in the user-application interaction, it is important to follow closely all the style rules dictated by CUA 91, so that all applications conform to what is a market standard. This might seem unimportant, especially when thinking about a program's quality in absolute terms; however, it is appreciated by most nonprofessional computer users. Some style rules were implied in the previous paragraphs. Now we will have a look at all rules in an orderly manner.

The menu bar must always be alone and be laid out on one single line, even though multiple options might allow the implementation of hybrid forms, like horizontal menus or menus spanning several columns. In the menu bar, there should be top-level menus only, each featuring just one single text string (or, more rarely, one bitmap) with the first letter capital. Top-level menu must not have any kind of accessory element, like ellipses or exclamation marks, and it must always be possible to select them. Furthermore, top-level menus always fall in the *commands* category, and they originate a drop-down menu, or sometimes perform an action immediately. (The menu bar can appear on multiple lines only if the width of the window is limited and insufficient to accommodate all top-level options on one line.)

If a top-level menu includes a drop-down menu, this will be declared with the `SUBMENU` directive, and will not produce any kind of direct action with regards to the application. On the other hand, if the selection of a top-level menu causes an immediate action from part of the program, then this will have been declared a `MENUITEM`. As we will see in the next few paragraphs, the selection of a `MENUITEM` will always cause a message to be sent to the window procedure of the client area with which the menu bar is associated. In the case of a top-level `SUBMENU`, its selection will simply cause the associated drop-down menu to be displayed, but no selection message is ever sent to the client area's window procedure.

Drop-down menus must be organized in such a way as to present just one `MENUITEM` per line, possibly grouped and delimited by a horizontal line. The subdivisions are useful for defining blocks of related options, and will make it easier for the user to find the options needed. The text string that characterizes each `MENUITEM` should not be excessively long, so that overly large portions of the application's client area are not hidden from sight.

The definition of a `SUBMENU` sprouting from a previous `SUBMENU` is a typical feature of a `MENU` resource, although it should not be overused. In fact, the presence of several concatenated `SUBMENUS` will usually make it harder rather than easier for the user to make a selection, as the application becomes impaired by an overburdened interface. The maximum number of concatenated `SUBMENUS` that can reasonably be accepted is two. This is a rule that will be followed in all the samples presented here and in the menu generator furnished in this text and described further ahead in Chapter 9.

Furthermore, the CUA 91 specifications are uncompromising on this issue, and define the technique for implementing multiple cascading drop-down menus as "... a solution valid only when no other technique can be adopted." In practice, it is advisable not to define any drop-down menus beyond the first level. If you really can't do without them, keep to second level drop-down conditional menus.

For any `MENUITEM`, either top-level or part of a `SUBMENU`, it is necessary to pick a letter as the mnemonic for keyboard selection. The definition of a unique letter pertains to all top-level menus, and all options within a drop-down menu. It is possible, however, to use the same letter adopted in a top-level menu inside its associated drop-down menu.

When defining accelerators, it is important to follow the standards set by the major commercial applications, and follow the conventions used by them (Ctrl+Ins, for instance) and avoid combinations of two or more keys.

---

## Loading a Menu Template

It is necessary to make some changes to the application's source file in order to exploit the a menu resource described in the RC file. First, you have to insert the `#include "menuapp.h"` directive so that the code can get to know the IDs assigned to the `MENUITEM` in the menu template.

There are several ways to load a menu template. The most straightforward is to take advantage of the services provided by *WinCreateStdWindow()*. First, get rid of any operation setting the FCF\_MENU flag if you ever defined the FCF\_STANDARD style as a starting point. (This is what has been done in all previous examples.) The syntax of *WinCreateStdWindow()* for a window associated with a menu bar expects as its eighth parameter, the ID of the resource, that is, RS\_MENU in Listing 6.1 or RS\_ALL if you also have an icon. .

A second method is to handle the situation as described in Chapter 5: RS\_ALL is the ID assigned to the menu, the icon, and the accelerator table (which is absent for now) in the resource file. By indicating RS\_ALL, you instruct *WinCreateStdWindow()* to load the resources indicated by the FCF\_ flags and identified by that ID (the MENU and the ICON resource in our example). Listing 6.3 contains the MENUAPP application's code, showing a menu window, as you can see in Figure 6.11.

In Listing 6.3 you can see some new PM programming elements. The resource file contains the definition of the icon to be associated with the application's window. The ID assigned to this resource is the same one specified in the MENU directive. In fact, in the source code the FCF\_ flags specified are all those provided by FCF\_STANDARD, with the exception of FCF\_ACCELTABLE. This means that when the window is created with *WinCreateStdWindow()* the icon MENUAPP.ICO will be assigned in addition to the menu template.

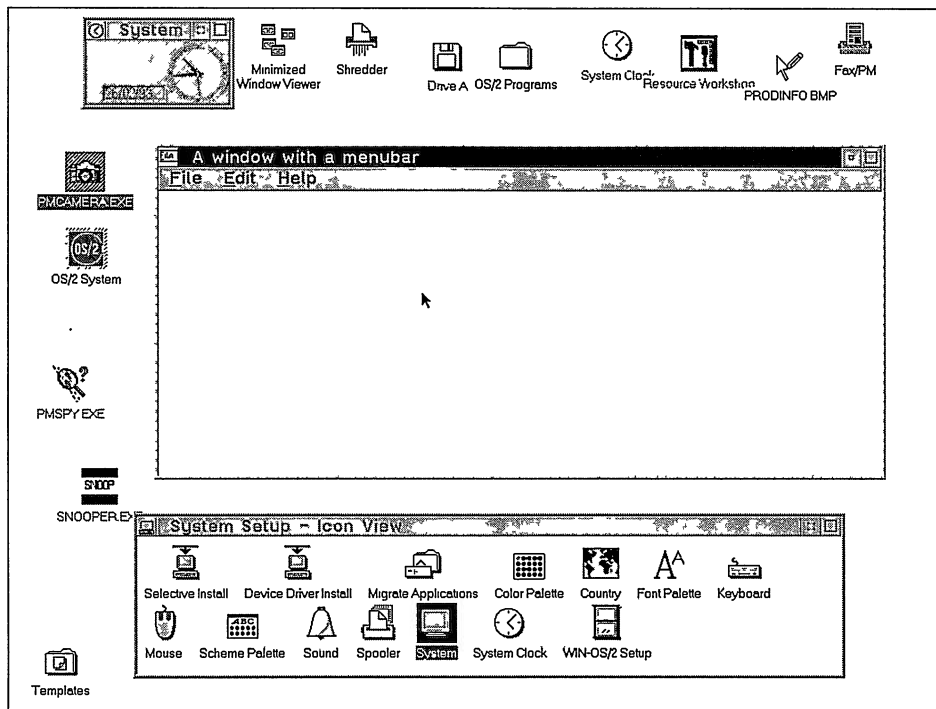


Figure 6.11 The MENUAPP application has a menu bar.

Finally, the resource file also contains a STRINGTABLE resource with some text strings used by the application. In the code you can see that *WinLoadString()* is called to load the class name before registering it and before creating the window:

```
// MENUAPP.C
#include "menuapp.h"
...
CHAR szClassName[ 15] ;
CHAR szWindowTitle[ 15] ;
ULONG flFrameFlags = FCF_STANDARD & ~FCF_ACCELTABLE ;
...
// load class name from resource file
WinLoadString( hab, NULLHANDLE,
               ST_CLASSNAME,
               sizeof( szClassName), szClassName) ;
WinLoadString( hab, NULLHANDLE,
               ST_WINDOWTITLE,
               sizeof( szWindowTitle), szWindowTitle) ;

// register class
WinRegisterClass( hab, szClassName,
                  ClientWndProc,
                  CS_SIZEREDRAW, 0L) ;

// create main window
hwndFrame = WinCreateStdWindow( HwndDesktop,
                                WS_VISIBLE,
                                &flFrameFlags,
                                szClassName,
                                szWindowTitle,
                                0L,
                                NULLHANDLE,
                                RS_ALL,
                                &hwndClient) ;
...
```

The set of windows controlled by the frame window is enlarged with another element: the menu bar. It is an ordinary PM window belonging to the WC\_MENU class, characterized by an appropriate handle that allows you to identify and use it, even though it has been created independently by *WinCreateStdWindow()*. To access the handle, call the *WinWindowFromID()* function and pass it the FID\_MENU ID:

```
Hwnd hmenu ;
...
hmenu = WinWindowFromID( hwndFrame, FID_MENU) ;
...
```

Once the menu template has been created and loaded indirectly via *WinCreateStdWindow()*, it is possible to interact with the new window, and the application will be aware of the selections performed by the user from the various top-level and drop-down menus.

If, instead, you wish to operate independently from *WinCreateStdWindow()*, then you must first use the function *WinLoadMenu()* or *WinCreateMenu()*. While the first one will essentially read a menu template and then transform that information into a window belonging to the *WC\_MENU* class, the second one will also return a handle to a menu, a menu that, however, does not exist and that will have to be created from scratch.

```
#define INCL_WINMENUMS
HWND APIENTRY WinCreateMenu( HWND hwndParent, PVOID pvmt ) ;
```

<i>Parameter</i>	<i>Description</i>
hwndParent	Handle to the parent window with which the menu is associated
pvmt	Pointer to a memory area containing the information regarding a menu template
<i>Return Value</i>	<i>Description</i>
HWND	Handle of a <i>WC_MENU</i> class window or <i>NULLHANDLE</i> in case of error

The first parameter identifies both the parent as well as the owner of the *WC\_MENU* class window created by this function. The second parameter is a binary format menu template, that is, the format it is transformed into by the resource compiler. As describing a menu through a template is so convenient, this second solution is rarely used.

---

## Menus, Parenthood, and Ownership

You must be very careful with the first parameter. The menu bar is a child window of the frame, identified through the ID *FID\_MENU*. Although the results produced by a user's selecting the various menu items available, the frame window is the owner and parent of the menu bar.

The return value of this function is the handle to the new menu window. (In PM, different from what happens in MS Windows, there is no specific type of handle for menus, and thus you simply use a *HWND*.)

All drop-down menus are child windows of the system's desktop, a window belonging to the class #32776 and is invisible to the user because it is covered by WPS. This is only half the picture. Drop-down menus are in fact a very interesting case of windows with interchangeable parents. Up to this point, you have been accustomed to thinking about the parent-child relationship as something that is static and unchangeable. In PM, different from what happens with human beings, it is possible to change parents simply by calling *WinSetParent()* described in Chapter 4. The drop-

down parent is `HWND_DESKTOP` when they are visible, and `HWND_OBJECT` for the rest of the time. `HWND_OBJECT` is equivalent to `HWND_DESKTOP` as far as the permanently invisible windows' hierarchy tree is concerned. The drop-downs will always have as their owner the menu bar. Figure 6.12 summarizes all these relationships.

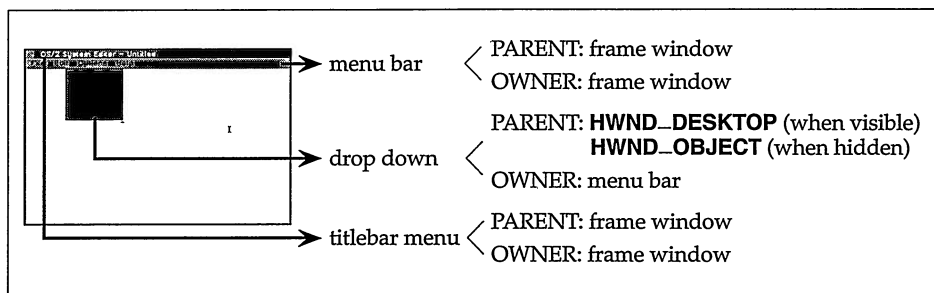
The client window has no relationship, neither parenthood nor ownership, with the menus. The selection of a top-level menu by pressing the left mouse button will cause the corresponding drop-down menu to be displayed, and thus will cause its parent to change from the object window to the desktop. It is not possible to determine in absolute terms which pixels will be covered by a drop-down menu when it is displayed. Most often, it will be a portion of the client window, but sometimes the vertical space available in a client window is not enough. For this reason, the parent of a drop-down menu is the desktop window (`HWND_DESKTOP`), apart from the vertical direction in which the window is displayed. Figure 6.13 displays the parenthood relationship of a drop-down menu.

## Modifying the Window Procedure

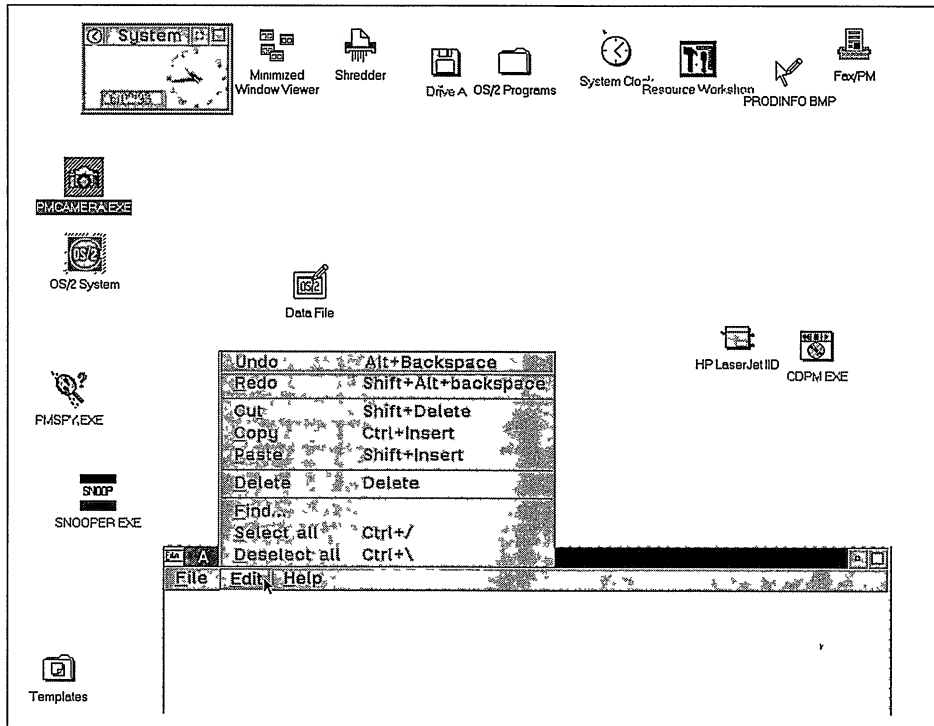
The selection of any kind of menu item—inside a `SUBMENU` or outside—will cause the `WM_COMMAND` message to be sent to the window procedure of the client window with which the menu window is associated (both windows are children of one same frame window, and are thus sibling windows). The space taken by the menu bar is obtained from the window by consuming pixels from the frame and reducing the height of the client window by the amount of pixels returned by the function:

```
lHeight = WinQuerySysValue( HWND_DESKTOP, SV_CYMENU );
```

The generation of the `WM_COMMAND` message addressed to the client window procedure takes place only for those menu items expressly declared with the `MENUITEM` directive in the resource file, or in some equivalent template image constructed in memory. Any other specimen of a `WC_MENU` class window of the `SUBMENU` type (thus top-level menus or those inside a drop-down menu), will not send a `WM_COMMAND`



**Figure 6.12** Relationship existing between the window of `WC_MENU` class, the frame, and the client window.



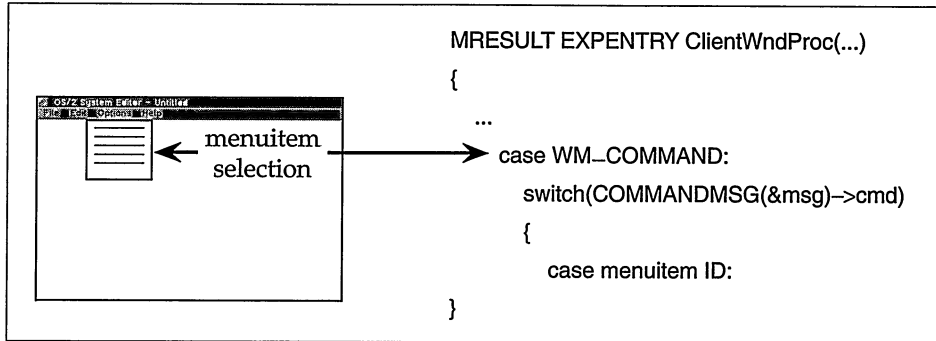
**Figure 6.13** A drop-down is displayed upward if the pixels available below the menu bar are not enough.

message to the application's window procedure, but will only change the visibility state of the associated drop-down. There exists an interaction between the client area and a top-level menu, represented by the receipt of the `WM_INITMENU` message. This is a kind of initialization message that is sent as a consequence of the focus being transferred to the menu bar. Catching the `WM_INITMENU` message is useful also for setting up dynamic changes to the contents of any associated drop-down. `WM_INITMENU` will not be issued by selecting a top-level `MENUITEM`.

On the basis of these considerations, it is clear that the window procedure of an application geared with a menu bar will receive a `WM_COMMAND` message for handling the selections performed by the user. Figure 6.14 summarizes the relationship that exists between the menu bar and the internal structure of a PM application.

The message `WM_COMMAND` contains in `mp1` the ID of the selected `MENUITEM`. In the window procedure it will be possible to identify the origin of `WM_COMMAND` by looking at `mp1`. PM's API provides the software designer with the `COMMANDMSG` macro, which makes it easy to extract the selected `MENUITEM`'s ID from `mp1`, and the information about the source of the message (keyboard, mouse, or accelerator) from `mp2` by acting directly on the `CMDMSG` structure:





**Figure 6.14** The selection of a MENUITEM will issue the WM\_COMMAND message to the window's window procedure.

```
#define INCL_WINMESSAGEGR
typedef struct _COMMANDMSG
{
    USHORT cmd ;    // mp1
    USHORT unused ;
    USHORT source ; // mp2
    USHORT fMouse ;
} CMDMSG ;
```

```
typedef CMDMSG *PCMDMSG ;
```

Here's the COMMANDMSG macro:

```
#define COMMANDMSG(pmsg) ((PCMDMSG)((PBYTE)pmsg + sizeof(MPARAM)))
```

The COMMANDMSG macro requires you to give it the address of the identifier that contains the message issued to the window procedure (the second parameter of any window procedure, generally called msg).

To get to know the ID and thus detect which menu item has been selected in the application's menu, you have to discover what value is present in the cmd member of CMDMSG, as you can see in the following code fragment:

```
...
case WM_COMMAND:
    switch( COMMANDMSG( &msg) -> cmd)
    {
        case MN_COPY:
            break ;

        case MN_UNDO:
            break ;

        case MN_CUT:
            break ;

        case MN_PASTE:
```

```

        break ;
    case MN_CLEAR:
        break ;
    case MN_NEW:
        break ;
    ...
    default:
        break ;
}
...
}
...

```

The case MN\_ conditions will need to be refined with specific code for implementing the application's functionality, considering the request formulated by the user through the selection. It is therefore possible that the code fragment dealing with a WM\_COMMAND message may be quite long. It is quite common to factor out from the window procedure all the logic dealing with menu handling by defining a stand-alone function that will be called immediately after the receipt of a WM\_COMMAND message, as in the following example:

```

...
case WM_COMMAND:
    DoCommands( hwnd, msg, mp1, mp2) ;
    break ;
...

```

where *DoCommands()* is a function developed along these lines:

```

void DoCommands(    HWND hwnd,
                   ULONG msg,
                   MPARAM mp1,
                   MPARAM mp2)
{
...
    switch( COMMANDMSG( &msg) -> cmd)
    {
        case MN_xxx:
            ... ;
            break ;
        ...
    }
    ...
}

```

Any subsequent addition of a new MENUITEM to a menu template will require that you implement a corresponding number of case branches inside the *DoCommands()* function.

In Listing 6.3 the `WM_COMMAND` message is processed in order to display in a message box the ID of the selected menu item, along with its corresponding text. The text of each menu item is retrieved directly from the drop-down to which it belongs. After the detection of `WM_COMMAND` the application sends a `MM_QUERYITEMTEXT` message to the action bar. To do this, it is necessary to have the menu's handle. You can get this information in many ways. In `MENUAPP`'s code you will find the two macros `PAPA` and `MENU` that serve this purpose by acting on the client window's handle, `hwnd`. It is also possible to intercept the `WM_INITMENU` message that will contain in `mp2` the handle of the drop-down associated with the selected top-level. Through this information it is easy to quickly get to the handle of the menu bar, as it is the owner.

To retrieve the text string associated with the selected menu item, you can try something new: Send a message to the menu window. This consists of setting up an addressee that is a window belonging to a predefined class. The message `MM_QUERYITEMTEXT` is specific for `WC_MENU` class windows, as you might infer from the `MM_` prefix, which stands for *menu message*. The complete list of messages for the `WC_MENU` class appears in Table 6.9.

The syntax of `MM_QUERYITEMTEXT` should have in `mp1` the ID of a menu item, and in `mp2` the address of a character array (a buffer) large enough to accommodate the menu item's text string.

		<i>Description</i>
<code>MM_QUERYITEMTEXT</code>	<code>0x018b</code>	
<code>mp1</code>	<code>USHORT usItem</code>	ID of the menu item
	<code>SHORT smaxcount</code>	Size of the character array in <code>mp2</code> .
<code>mp2</code>	<code>PSZ pItemText</code>	Array of characters containing the menu item's text
Return Value	<code>SHORT sTextLength</code>	Length of the text string or 0 in case of error

```

...
case WM_COMMAND:
    switch( COMMANDMSG( &msg) -> cmd )
    {
        case MN_COPY:
            ...
        case MN_PRODINFO:
            WinSendMessage( hwndMenu, MM_QUERYITEMTEXT,
                MPFROM2SHORT( COMMANDMSG( &msg) -> cmd,
                    sizeof( szBuffer),
                    MPFROMP( szBuffer)) );
            ...
    }
}

```

The selected option's text string is inserted into `szBuffer`, an array of `CHAR`. Naturally, the text string will also contain any accessory style elements, like an ellipsis, a tilde, or a tab character. The `StringParser()` function has been designed specifically for getting rid of all unwanted characters in the text of a menu item:

**Table 6.9 The MM\_ Messages That Govern the Interaction with the Menus of a PM Application**

<i>Message</i>	<i>Value</i>	<i>Description</i>
MM_INSERTITEM	0x0180	Inserts a new menu item.
MM_DELETEITEM	0x0181	Deletes a menu item.
MM_QUERYITEM	0x0182	Returns data about a menu item in a MENU-ITEM structure.
MM_SETITEM	0x0183	Sets the values from a MENUITEM structure.
MM_QUERYITEMCOUNT	0x0184	Returns the number of menu items.
MM_STARTMENU MODE	0x0185	Signals the start of a menu selection.
MM_ENDMENU MODE	0x0186	Signals the end of a menu selection.
MM_DISMISSMENU	0x0187	Hides from view a drop-down menu featuring the MIA_NODISMISS attribute (which no longer appears in PMWIN.H).
MM_REMOVEITEM	0x0188	Removes a menu item from the menu structure.
MM_SELECTITEM	0x0189	Selects a menu item.
MM_QUERYSELITEMID	0x018a	Returns the ID of the selected menu item.
MM_QUERYITEMTEXT	0x018b	Returns the text of a menu item.
MM_QUERYITEMTEXTLENGTH	0x018c	Returns the length of the text of a menu item.
MM_SETITEMHANDLE	0x018d	Associates a handle with a menu item, typically that of a bitmap.
MM_SETITEMTEXT	0x018e	Sets the text of a menu item.
MM_ITEMPOSITIONFROMID	0x018f	Returns the position of a menu item on the basis of the specified ID.
MM_ITEMIDFROMPOSITION	0x0190	Returns the ID of a menu item starting from the indicated position.
MM_QUERYITEMATTR	0x0191	Retrieves a menu item's attributes.
MM_SETITEMATTR	0x0192	Sets the attributes (MIA_) of a menu item.
MM_ISITEMVALID	0x0193	Tests if a menu item is selectable.
MM_QUERYITEMRECT	0x0194	Returns the menu item size in a RECTL structure.
MM_QUERYDEFAULTITEMID	0x0431	Returns the menu item ID of the selected item in a conditional cascade drop-down.
MM_SETDEFAULTITEMID	0x0432	Sets the menu item ID of the selected item in a conditional cascade drop-down.

```

...
StringParser( szBuffer) ;
...

```

The “cleaned-up” string will be left by the function in `szBuffer`. There’s not much left to do other than prepare the new text to be displayed with the `WinMessageBox()` function:

```

...
sprintf( szString, "Command: %s ID: %3d", szBuffer,
        COMMANDMSG( &msg) -> cmd) ;

WinMessageBox(  HWND_DESKTOP, hwnd,
                szBuffer,
                "Message Received",
                0, MB_OK) ;

break ;
...

```

All menu items described in the menu template will produce the same output: a message box that notifies the menu’s ID and text. The only exception is the Exit option. The effect produced by its selection is the posting of the `WM_QUIT` message into the application’s message queue. That will terminate its execution:

```

...
case MN_EXIT:
    WinPostMsg( hwnd, WM_QUIT, 0L, 0L) ;
    break ;
...

```

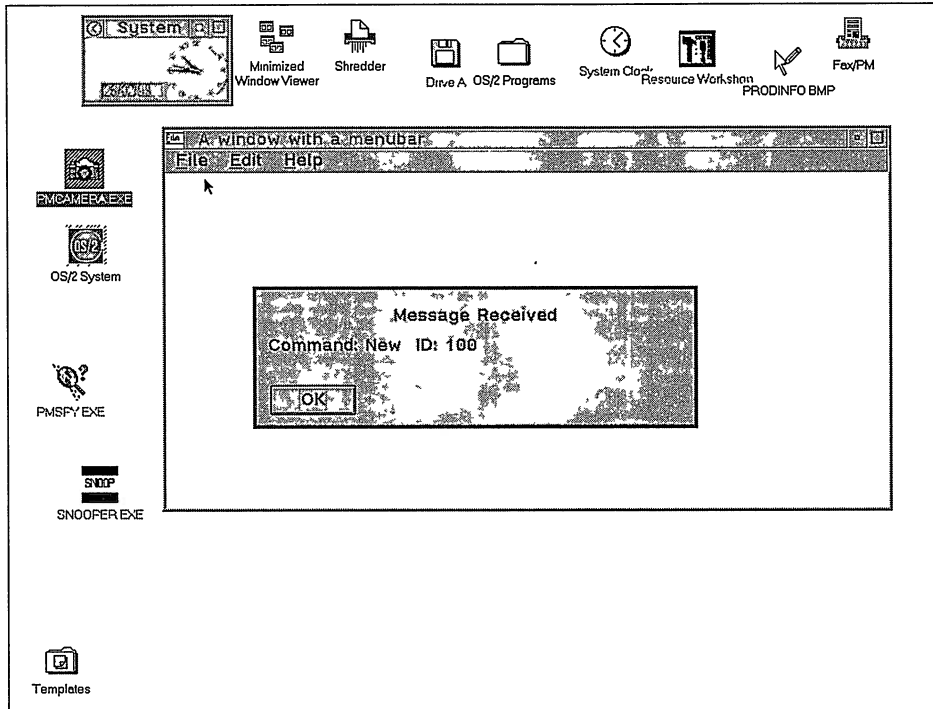
In general, there will be no default branch in the switch block dealing with a `WM_COMMAND` message, for the very reason that you intend to find all possible conditions described by the template. In the specific case of `MENUAPP` (Listing 6.3), it is possible to avoid to indicate the diverse conditions and replace them all with one simple default branch. In this way, all IDs with the exception of `MN_EXIT` will be subject to the same kind of processing from the application.

`MENUAPP` (Figure 6.15) can also be used to test the functioning of the ALT+mnemonic key combinations to select a top-level menu and then a menu item in the corresponding drop-down.

---

## Recognizing the Source of a Selection

In the next example you will try to determine the source that originated the selection of a `MENUITEM`. The possible sources are the keyboard, the mouse, and an accelerator combination (with the keyboard). To know about the source, you only need to extrapolate the



**Figure 6.15** When a menu item is selected, the application responds by displaying a message box.

information in `mp1` and `mp2` through the `COMMANDMSG` macro and compare the returned value with what is listed in Table 6.10.

The member `fMouse` of type `B00L` indicates whether the message is the consequence of some pointing action through the mouse (`TRUE`) or through the keyboard (`FALSE`). On the basis of this, it is possible to build an application that uses menu windows, and interacts with the user to detect the selection of a menu item. This data does not have any greater usefulness within an application (except, maybe, that of collecting statistics on the frequency with which accelerators are used compared to the mouse or the keyboard—and maybe allow a clever piece of software to configure itself on the basis of the user that is at the console and of the information previously collected). Listing 6.4 has been implemented along the same lines as the previous one, i.e., to retrieve information through the `COMMANDMSG` structure to detect the kind of input tool used to select a menu item.

Compared to Listing 6.3, the only notable change is that pertaining to the processing that takes place in the code fragment dealing with the `WM_COMMAND` message.

```
...
//determine source
switch( SHORT1FROMMP( mp2))
```

```

{
    case CMDSRC_ACCELERATOR:
        strcpy( szSource, "accelerator" ) ;
        break ;

    case CMDSRC_MENU:
        strcpy( szSource, "menu" ) ;
        break ;

    default:
        strcpy( szSource, "other" ) ;
        break ;
}
...

```

**Table 6.10 Values of the Source Member in mp2 According to the COMMANDMSG Structure**

<i>Define</i>	<i>Value</i>	<i>Description</i>
CMDSRC_PUSHBUTTON	0x0001	Indicates that the WM_COMMAND message has been sent to the application's window procedure as a consequence of the selection of a <i>pushbutton</i> . This condition has nothing to do with menu handling issues, but anticipates the fact the interaction between the user and a pushbutton—the windows belonging to the WC_BUTTON predefined class—get notified to the application through the same WC_COMMAND message.
CMDSRC_MENU	0x002	Indicates that the presence of a WM_COMMAND message in the application's window procedure is the consequence of the selection of one of the MENUITEMS that make up the application's menus.
CMDSRC_ACCELERATOR	0x003	The WM_COMMAND message is the consequence of the user pressing some keyboard accelerator (described in the next section).
CMDSRC_OTHER	0x000	The source of the WM_COMMAND message cannot be recognized precisely, probably because the message was generated by some other control or by the application itself.

With the SHORT1FROMMP macro, you can retrieve from mp2 the information identifying the input source that generated the WM\_COMMAND message. The conditions to consider are CMDSRC\_MENU and CMDSRC\_ACCELERATOR, in addition to a generic default branch. Now the output in the message box is more complete than in the previous example, as you can see in Figure 6.16.

## Changing Attributes Dynamically

The menu templates presented in the preceding paragraphs and in Listings 6.3 and 6.4 did not have any style or attributes (respectively MIS\_ and MIA\_). Often though, the customization of the menu items starts at the template level. Imagine, for example, that you are writing a template for a word processing application. The application's design does not expect any document to be loaded automatically, unless it is the result of an association among objects or of a drag & drop operation. The proposed scenario, thus, does not foresee the possibility of performing any storage operations with options like Save and Save as..., for the simple reason that at that moment there is nothing to save. Hence, it is convenient to set up a menu template that takes into consideration these design constraints, and assigns each menu item appropriate

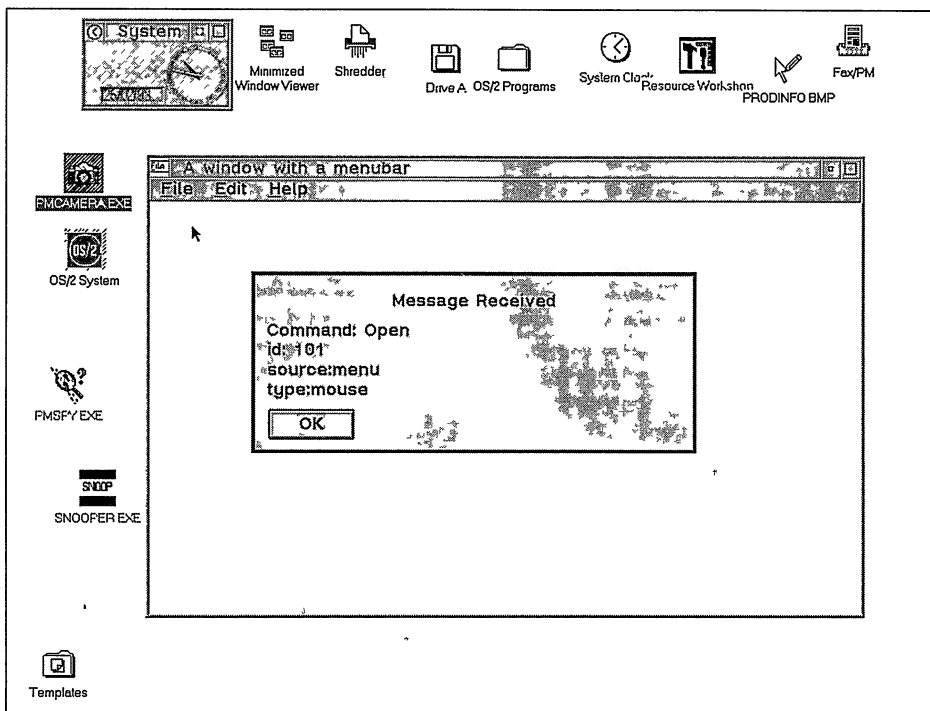


Figure 6.16 Output of the program in Listing 6.4, as it appears after selecting a menu item with the mouse.



attributes. You might want to use the `MIA_DISABLED` attribute to disallow the selection of a menu item that would issue the `WM_COMMAND` to the client's window procedure. The attributes have to appear after the styles; so the Save menu item will take on the following look:

```
MENUITEM "~Save", MN_SAVE, , MIA_DISABLED
```

The same logic seen for `MIA_DISABLED` can be applied to all other attributes. The Save option appears colored in gray inside the drop-down menu, and is unselectable. Its selection through the mouse or the keyboard will automatically emit a beep, and it will not issue a `WM_COMMAND` message. It makes sense that a potential user of our hypothetical word processor will sooner or later want to save an edited document. Therefore, the Save option must change its state during the application's execution, and must abandon its condition of being disabled. In order to do this, you must send the message `MM_SETITEMATTR` directly to the menu window, indicating the ID of the menu item involved in the change, in addition to the precise attributes needed.

After `MM_QUERYITEMTEXT`, `MM_SETITEMATTR` is a second message specific to `WC_MENU` windows.

	MM_SETITEMATTR	0x0192	<i>Description</i>
	<code>mp1</code>	<code>USHORT usItem</code>	ID of the menu item
		<code>USHORT usinclude-submenus</code>	Whether the search for the menu item will exclude the <code>SUB-MENU</code> s ( <code>FALSE</code> ) or it will include them ( <code>TRUE</code> )
	<code>mp2</code>	<code>USHORT usattributemask</code>	Attributes to consider
		<code>USHORT usattributedata</code>	State of attributes (set/cleared)
	Return Value	<code>BOOL fsuccess</code>	Success or failure of the operation

The logic governing these messages demands that the addressee always be a menu window. The designer can specify either the handle of the menu bar or the handle of a drop-down menu directly. This last piece of information is not accessible directly, but it can be retrieved by sending a `MM_QUERYITEM` message.

This means that the `MM_` messages are sent to the menu bar, which, being the owner of all drop-downs associated with the top-levels, will dispatch them directly to the item concerned (and this is yet another reason for which all IDs at the menu template level should be unique). This approach simplifies a great deal of the software designer's work. Once you get to know the handle of the menu bar and the IDs, it is possible to access any of the menu items present in the menu structure. The sending of the `MM_SETITEMATTR` message will always take place through the `WinSendMessage()` function, with the following syntax:

```
...
WinSendMessage( hmenu, MM_SETITEMATTR,
                MPFROM2SHORT( MN_SAVE, TRUE),
                MPFROM2SHORT( MIA_DISABLED, FALSE) );
...
```

The handle `hmenu` refers to the menu bar, which you know how to retrieve. The two packed `SHORT` in `mp1` identify, respectively, the menu item referred to by the message (`MN_SAVE`), and the way it should be searched for. By specifying `TRUE`, you inform PM that the search should extend to all drop-downs present in the menu structure. Since the use of attributes is limited by the style rules to menu items alone, this second parameter is almost always set to `TRUE`.

Somewhat more complex is the meaning of the two `SHORTs` in `mp2`. The first one specifies the attribute you are acting on (you could even indicate multiple attributes through a bitwise `OR`). The second one represents the new state. In the example, you want to inform PM that the attribute `MIA_DISABLED` that is present in the `MN_SAVE` menu item has to be cleared. Another alternative and better solution could be the following one:

```
...
WinSendMsg( hmenu, MM_SETITEMATTR,
            MPFROM2SHORT( MN_SAVE, TRUE),
            MPFROM2SHORT( MIA_DISABLED, ~MIA_DISABLED) );
...
```



calling for the removal of the bit corresponding to the `MIA_DISABLED` attribute. Listing 6.5 is an evolution of the preceding ones. The menu items `Save` and `Save As...` are disabled directly in the template residing in the resource file. The user will be able to select them only after loading a document (in the example, you will only have to select the `Open` option from the `File` menu). Furthermore, the menu item `New` will initially have a check-mark symbol. Subsequent selections will modify this attribute, and implement a behavior typical of a state setter. (Note that the setting of the `MIA_CHECKED` attribute for the `New` option is not compliant with CUA specifications, but is useful for the learning purpose of this example.)

The `New` option must display a check-mark symbol if it did not have one in its former state. The logic that governs the processing of this ID must first evaluate the state of the corresponding menu item, with respect to the `MIA_CHECKED` attribute. The message `MM_QUERYITEMATTR` will return a boolean value reflecting the state of the appropriate attribute (or attributes). On the basis of the value returned by this query, you can go ahead and send the `MM_SETITEMATTR` message indicating that the attribute should be set or cleared, whichever is the opposite of the current condition.

```
...
case MN_NEW:
{
    BOOL fValue ;
    fValue = !WinSendMsg( hmenu, MM_QUERYITEMATTR,
                        MPFROM2SHORT( MN_NEW, TRUE),
                        MPFROMSHORT( MIA_CHECKED) );
    WinSendMsg( hmenu, MM_SETITEMATTR,
                MPFROM2SHORT( MN_NEW, TRUE),
                MPFROM2SHORT( MIA_CHECKED,
                            (fValue) ? MIA_CHECKED : ~MIA_CHECKED) );
}
break ;
...
```

When, instead, the user selects the Open menu item, you have to enable the Save and Save As... options—simulating an effective loading of a document in the hypothetical word processor.

```

...
case MN_OPEN:
    WinSendMsg( hmenu, MM_SETITEMATTR,
                MPFROM2SHORT( MN_SAVE, TRUE),
                MPFROM2SHORT( MIA_DISABLED, ~MIA_DISABLED)) ;

    WinSendMsg( hmenu, MM_SETITEMATTR,
                MPFROM2SHORT( MN_SAVEAS, TRUE),
                MPFROM2SHORT( MIA_DISABLED, ~MIA_DISABLED)) ;

break ;
...

```

The operation is performed for every selection of Open; naturally, the code should also provide somewhere the condition that disables the Save and Save As... options if the word processor no longer has open documents that can be saved.

---

## Messages and Macros

In the preceding examples, you have seen a certain level of interaction between the application (the client's window procedure) and the windows of the WC\_MENU class, by means of sending MM\_ messages. This is the only solution available. However, to make the software designer's life a little easier and to make the program listing somewhat simpler, starting with version 2.0 of OS/2 some handy macros have appeared. They are a series of defines that encapsulate what looks like a function by the sending of particular MM\_ message to a WC\_MENU class window. The advantage offered by this solution is that of avoiding part of the syntax of each message, leaving it up to the macro to define some of the parameters. The menu macros are listed in Table 6.11.

One of the most promising aspects of these macros is that they are documented in the online help system. To learn about their syntax, just invoke the help system, just as for any other function (the Win prefix emphasizes this aspect even more). In the listings presented in this chapter, we have always used the syntax based on an explicit *WinSendMsg()*, for instructive purposes.

---

## Loading a New Menu

When you develop complex PM applications, a solution embraced by some software houses is that of following the *Multiple Document Interface* development model. The structure of a MDI application allows for multiple child windows to be displayed in the client area of a top-level window. Lotus 1-2-3 for OS/2 2.1 is a typical representative of this category of applications. In 1-2-3 there are several different kinds of document windows: worksheets, macro sheets, and graph sheets. Each document

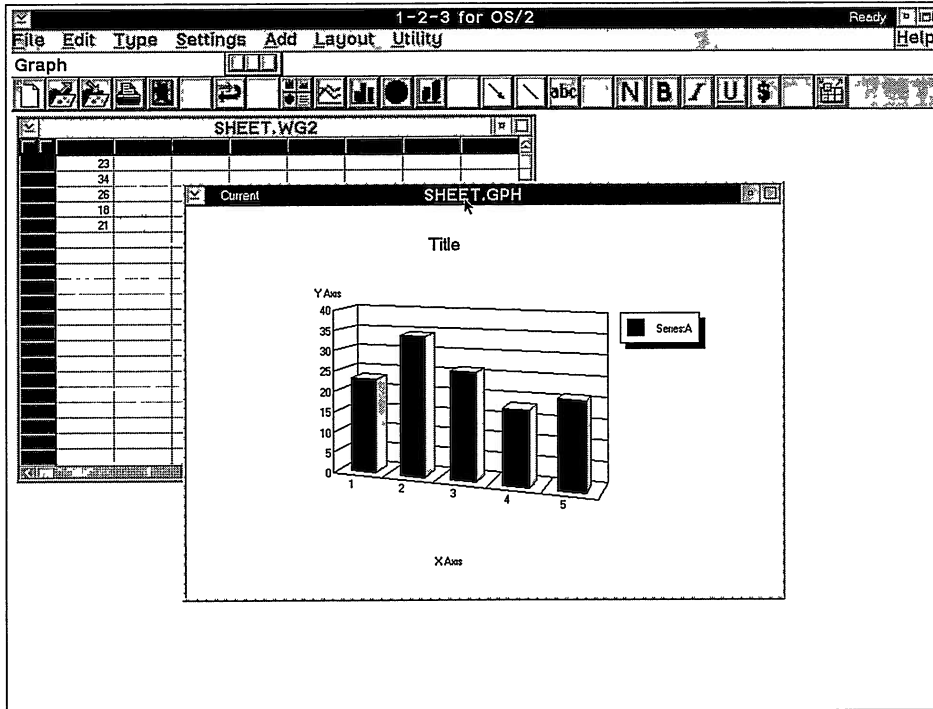
**Table 6.11 The Macros Present in PMWIN.H That Simplify Some of the Operations Regarding Interaction with WC\_MENU Class Windows**

<i>Macro</i>	<i>Description</i>
WinCheckMenuItem (hwndMenu, id, fcheck)	Sets/clears the MIA_CHECKED attribute for the ID menu item by sending the MM_SETITEMATTR message.
WinIsMenuItemChecked (hwndMenu, id)	States whether the ID menu item is displayed together with a check-mark symbol (MM_QUERYITEMATTR).
WinEnableMenuItem (hwndMenu, id, fEnable)	Enables/disables the ID menu item by setting/ clearing the MIA_DISABLE attribute through the MM_SETITEMATTR message.
WinIsMenuItemEnabled (hwndMenu, id)	States whether the ID menu item is enabled/disabled (MM_QUERYITEMATTR).
WinSetMenuItemText (hwndMenu, id, psz)	Sets the text of the ID menu item (MM_SETITEMTEXT).
WinIsMenuItemValid (hwndMenu, id)	States whether the ID menu item is valid (MM_ISITEMVALID).

window is lacking a menu bar (according to CUA specifications), and resorts to the main windows menu bar. In the case of work sheets and macro sheets, the options shown in the menu bar and in the associated drop-down menu will satisfy all user requirements. For graph sheets, there will be some different options, such as those for selecting the type of graph. Lotus 1-2-3 responds to this need by loading a second, different menu template into the application as soon as focus is given to a graph document. If focus is later transferred to a work sheet or a macro sheet, the original menu is restored (Figure 6.17).

There are several ways to accomplish this. The simplest is to generate inside the resource file two distinct menu templates that are selectively loaded into the program according to the document window that is active at any given moment (by intercepting the WM\_ACTIVATE message in the window procedure of the document classes). It is also possible to destroy or remove one or more top-level menus (and thus their associated drop-downs), and replace them with new ones for the new working conditions. In this case, it will be necessary to write more code for the application, and, above all, you will have to face some formidable problems, because menu handling is not as clean in PM as it could be.

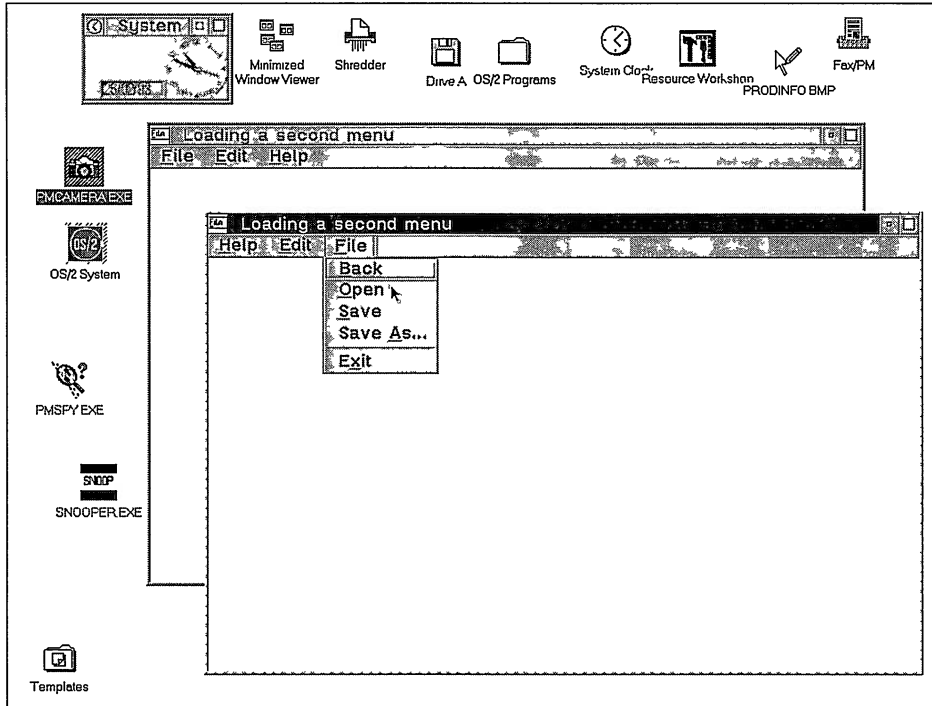
The two menu templates defined in the resource file have some similarities, as not all options need to be changed. The menu items that are common to the two templates share the same IDs, since they will be subject to exactly the same kind of processing. In this case, selecting different IDs would not hold any advantage, but would just unnecessarily burden the application. When the program's main window is displayed, a menu has to be loaded. The operation is always performed by calling *WinCreateStdWindow()*, being careful that the resource has the same ID as the



**Figure 6.17** The contents of Lotus 1-2-3's menu bar changes according to the active document window.

window's icon, and maybe even as the accelerator table (if any). This template is referred to as the *main template*. When the "triggering" event happens, you will have to destroy the menu that is displayed at that moment, load the second one, and display it. Listing 6.6 represents the application TWOMENUS equipped with two menus in the resource file. The second one is replaced by the first one when the user selects the option New from the menu File. To restore the window to its starting condition, just select the option Back from the menu File. The biggest difference between the two menus is the order of the top-level menus: File, Edit, and Help for the first one; and Help, Edit, and File for the second. Figure 6.18 shows two different executions of TWOMENUS, each one featuring a different menu bar.

The menu bar handle is stored in the static class identifier `hmenu` by intercepting `WM_INITMENU`. This application's governing logic is concentrated in the case branch dealing with the New menu item's processing. The destruction of the menu window is performed with `WinDestroyWindow()`. The destruction will make the menu bar disappear from the window, then the new menu is loaded through `WinLoadMenu()`. The operation collapses into one call both the actual loading and the association of the menu with the application window, since the first parameter of the function is passed by the handle of the frame window. There is nothing else left to do other than to display



**Figure 6.18** The same window shows in the first case the starting menu, and, below, the menu loaded after the selection of the option New.

the new menu by forcing a refresh of the application's frame by sending the message WM\_UPDATEFRAME.

```

...
case MN_NEW:
    WinDestroyWindow( hmenu ) ;
    hmenu = WinLoadMenu( PAPA( hwnd ), NULLHANDLE, RS_SECOND ) ;
    WinSendMessage( PAPA( hwnd ), WM_UPDATEFRAME,
        MPFROMSHORT( FCF_MENU ), 0L ) ;
    break ;
...

```



Listing 6.7 reports a modified version of TWOMENUS. In this case the handles of the two menus are stored in two different static class identifiers, directly in the code of WM\_CREATE: in such a way, they are always available in the program. Of these two menus, the first one is loaded when the application's window is created. For the second one, instead, special handling is used. During the processing of the WM\_CREATE message the second menu is loaded with *WinLoadMenu()*, specifying *HWND\_OBJECT* as its parent window. Before terminating the processing of the WM\_CREATE message, it is necessary to set the owner of the second menu; that is done by referencing the application's frame window with *WinSetOwner()*.

```

...
case WM_CREATE:
    hmenu1 = MENU( PAPA( hwnd)) ;
    hmenu2 = WinLoadMenu( HWND_OBJECT, 0, RS_SECOND) ;
    WinSetOwner( hmenu2, PAPA( hwnd)) ;
    break ;
...

```

When the application receives the `WM_COMMAND` message regarding the `MN_NEW` menu item (requesting the switch to the second menu), it only needs to exchange the parents of the two menu windows. The main menu acquires as its parent `HWND_OBJECT`, while the second one will get as its new parent the application's frame window. To complete the exchange between the two menus, you must force the window's output to update by sending the `WM_UPDATEFRAME` message.

```

...
case MN_NEW
    WinSetParent( hmenu1, HWND_OBJECT, FALSE) ;
    WinSetParent( hmenu2, PAPA( hwnd), TRUE) ;
    WindSendMsg( PAPA( hwnd), WM_UPDATEFRAME,
        MPFROMSHORT( FCF_MENU), 0L) ;

    break ;
...

```

---

## Bitmaps As Menu Items

Menu items are not limited to text strings alone. A bitmap can play the same role, and does not impose any harder work on the software designer. Inserting a bitmap in a menu item is a very easy operation in PM, both in the resource file as well as during program execution. Let's start with the changes necessary in the resource file's menu template. Each menu item made up of a bitmap will have the special `MIS_BITMAP` style set, in order to be distinguished from the default `MIS_TEXT` style. In place of the text string enclosed in double-quotes, you will have the following notation:

```
#bitmapID
```

where `bitmapID` corresponds to the ID assigned to a bitmap—a BMP file previously indicated in a `BITMAP` directive. Therefore, the complete syntax requires you to declare some bitmaps earlier in the resource file, and then use those IDs for each bitmapped menu item.

```

...
BITMAP 1 RED.BMP
...
MENU _RS_MENU
{
    SUBMENU "~Colors", MN_COLORS
    {

```

```

    MENUITEM "#1", MN_RED, MIS_BITMAP
    ...
}
...
}

```

The source code will not require any modification as far as message handling of menu item selection is concerned, even if the menu item is a bitmap. The window procedure will always receive the WM\_COMMAND message containing the ID of the selected menu item. Listing 6.8 presents the BMPMENU application illustrated in Figure 6.19. As for text menu items, the size of the drop-down menu will be determined by the largest menu item.

The selection of this menu item will cause the client area to be colored the same color as the bitmap.

---

## Menus Built by the Application

Menu items can be text strings or bitmaps (maybe more colorful and pleasant to look at than those in Listing 6.8). This is not the whole story. PM's API function set also allows you to delegate to the application the task of handling the generation and

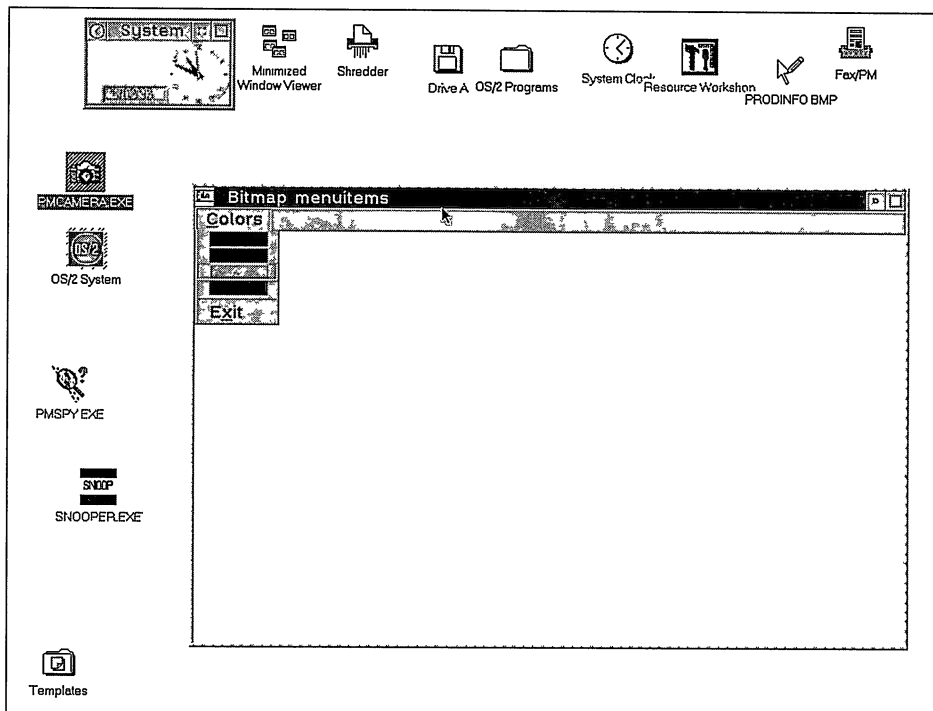


Figure 6.19 Structure of a drop-down menu containing bitmaps.



output of menu items. This will generate *owner-draw*-type menu items, which means that the task of controlling all painting operations is delegated to the window's owner. The basic requirement is setting the `MIS_OWNERDRAW` style for each menu item that is not to comply with the rules for the `WC_MENU` class for drawing its output; instead it will defer the whole operation to the application's internal logic.

Why would you want to create menu items whose output is controlled directly by the application's code? The answer is simple. Menus are windows belonging to the `WC_MENU` class, and, as such, they are completely external to the code written by the programmer. The only operations that are allowed are a certain degree of customization through styles and attributes (`MIS_` and `MIA_`), and a limited interaction through the sending of `MM_` prefixed messages. If the programmer intends to overcome the standard operating limits of menu items, there is no other way than that of directly handling the menu's output.

For example, the check-mark symbol cannot be changed through API services. If ever you had to use a different kind of symbol, you would have no other choice but to set the `MIS_OWNERDRAW` style for that particular menu item and set up the application (in the window procedure of the class to which the client area belongs) by writing the appropriate code to handle the situation. In practice, the assignment of `MIS_OWNERDRAW` is interpreted by the `WC_MENU` class's window procedure as an order to call some external procedure capable of processing the `WM_PAINT` message. The transfer of execution is accomplished by sending the `WM_DRAWITEM` message, which contains all information necessary for drawing in a window, that otherwise would be almost impossible. Since the governing rules for implementing owner-draw windows are identical for all predefined window classes that support this feature, we will defer treatment of the subject to Chapter 7.

---

## Accelerators

The Edit menu presented in the previous listings always had some text—even in the column reserved for accelerators (Figure 6.9). For the moment, these strings have not been associated with any of the application's resources. To make them work, you have to define an accelerator table in the RC file. An *accelerator* is a combination of one or more (generally two) keys, which correspond to the selection of a menu item. The PM application model provides for this additional selection tool in order to make operations quicker for experienced users. In fact, pressing a pair of keys is often much faster and easier than performing a manual or mouse selection of a certain menu item.

Not all menu items will have equivalent accelerators: Only those options that are important in the application's logic should have accelerators—for instance, those options dealing with cut and paste operations, like Cut, Delete, Copy, and Paste on the Edit menu.

The string `Ctrl+Ins` that appears to the right in the Copy menu item informs the user that this alternative keyboard combination is available to perform a copy of the object selected earlier. To make this keyboard combination operative, you must create an `ACCELTABLE` resource in the resource file, along the following syntax:

```
ACCELTABLE accelID [load options][memory management options]
{
    ...
}
```

ACCELTABLE, like many other directives we have already seen for the resource compiler, requires an ID and some optional load and memory management options. The ACCELTABLE block will accommodate all key combinations that are to be equated to the manual selection of a menu item. These statements consist of four parts:

```
key, menuItemID, key type, options
```

The key item references the various possible combinations. Most often, it will be the define of a virtual key, as it is defined in PMWIN.H. At other times, it will be a single key, or a symbol enclosed in double quotes, or even the ASCII code of a symbol. After this element you must put the ID assigned to the menu item to which the accelerator should refer. The key type is a piece of information allowing the resource compiler to recognize the kind of key combination. The accelerator declaration is completed by one or more options to indicate potential additional keys. Let's examine the Edit drop-down menu as it has been defined in the earlier examples.

```
...
SUBMENU "~Edit" , MN_EDIT
{
    MENUITEM "~Undo\tAlt+Backspace", MN_UNDO,, MIA_DISABLED
    MENUITEM "~Redo\tShift+Alt+backspace", MN_REDO,, MIA_DISABLED
    MENUITEM SEPARATOR
    MENUITEM "Cu~t\tShift+Delete", MN_CUT,, MIA_DISABLED
    MENUITEM "~Copy\tCtrl+Insert", MN_COPY,, MIA_DISABLED
    MENUITEM "~Paste\tShift+Insert", MN_PASTE,, MIA_DISABLED
    MENUITEM SEPARATOR
    MENUITEM "~Delete\tDelete", MN_DELETE,, MIA_DISABLED
    MENUITEM SEPARATOR
    MENUITEM "~Find...", MN_FIND,, MIA_DISABLED
    MENUITEM "~Select all\tCtrl+/", MN_SELECTALL,, MIA_DISABLED
    MENUITEM "~Deselect all\tCtrl+\\", MN_DESELECTALL,, MIA_DISABLED
}
...
```

The accelerator needed for the Undo menu item corresponds to the pressing of the ALT and BACKSPACE keys. The accelerator table must look like this:

```
ACCELTABLE RS_ACCELTABLE
{
    VK_BACKSPACE, MN_UNDO, VIRTUALKEY, ALT
    ...
}
```

The Backspace key does not have a proper ASCII code and is described by the VK\_BACKSPACE define in PMWIN.H. MN\_UNDO is the ID of the Undo menu item. All this means that the key combination ALT+BACKSPACE will cause the WM\_COMMAND message to be sent to the application, with MN\_UNDO as the command that is buried in

the CMDMSG structure. The options VIRTUALKEY and ALT inform the resource compiler that the first definition—a number—refers to a virtual key defined in PMWIN.H, while ALT is a key that has to be pressed together with BACKSPACE in order to obtain the desired result.

Let's now complete the accelerator table for the menu Edit, in accordance with the rules just described.

```

...
ACCELTABLE RS_ACCELTABLE
{
    VK_BACKSPACE,    MN_UNDO,          VIRTUALKEY, ALT
    VK_BACKSPACE,    MN_REDO,         VIRTUALKEY, ALT, SHIFT
    VK_DELETE,       MN_CUT,          VIRTUALKEY, SHIFT
    VK_INSERT,       MN_COPY,         VIRTUALKEY, CONTROL
    VK_INSERT,       MN_PASTE,        VIRTUALKEY, SHIFT
    VK_DELETE,       MN_DELETE,       VIRTUALKEY
    "/",             MN_SELECTALL,   CHAR, CONTROL
    "\\ ",           MN_DESELECTALL, CHAR, CONTROL
}
...

```

Table 6.12 summarizes the parameters that can qualify an ACCELTABLE resource in an RC file. CHAR, SCANCODE, and VIRTUALKEY are mutually exclusive, as are SYSCOMMAND and HELP.

Table 6.13 lists all virtual keys as defined in PMWIN.H.

**Table 6.12 Items That Characterize the Declaration of an Accelerator**

<i>Parameter</i>	<i>Description</i>
CHAR	The first item specified in an ACCELTABLE definition refers to a character in the ASCII set.
SCANCODE	The first item specified in an ACCELTABLE definition refers to a keyboard scancode.
VIRTUALKEY	The first item specified in an ACCELTABLE definition is a virtual key defined in PMWIN.H.
SHIFT	The accelerator key must be pressed together with one of the two SHIFT keys.
CONTROL	The accelerator key must be pressed together with the Ctrl key.
ALT	The accelerator must be pressed together with the Alt key.
SYSCOMMAND	Striking the accelerator combination issues the WM_SYSCOMMAND message in place of the default WM_COMMAND message.
HELP	Striking the accelerator combination issues the WM_HELP message in place of the default WM_COMMAND message.
LONEKEY	Indicates that the first item in the declaration of the accelerator (the key) can be pressed alone.

**Table 6.13 All Virtual Keys Defined in PMWIN.H**

<i>Key</i>	<i>Value</i>	<i>Description</i>
VK_BUTTON1	0x01	Mouse button 1
VK_BUTTON2	0x02	Mouse button 2
VK_BUTTON3	0x03	Mouse button 3
VK_BREAK	0x04	Break key
VK_BACKSPACE	0x05	Backspace key
VK_TAB	0x06	Tab key
VK_BACKTAB	0x07	Backtab key
VK_NEWLINE	0x08	New line key
VK_SHIFT	0x09	Shift key
VK_CTRL	0x0A	Control key
VK_ALT	0x0B	ALT key
VK_ALTGRAF	0x0C	ALT GRAF key (on European keyboards)
VK_PAUSE	0x0D	Pause key
VK_CAPSLOCK	0x0E	Caps Lock key
VK_ESC	0x0F	Escape
VK_SPACE	0x10	Space bar
VK_PAGEUP	0x11	Page up
VK_PAGEDOWN	0x12	Page down
VK_END	0x13	End
VK_HOME	0x14	Home
VK_LEFT	0x15	Left cursor key
VK_UP	0x16	Up cursor key
VK_RIGHT	0x17	Right cursor key
VK_DOWN	0x18	Down cursor key
VK_PRINTSCRN	0x19	Print screen
VK_INSERT	0x1A	Insert
VK_DELETE	0x1B	Delete
VK_SCROLLLOCK	0x1C	Scroll lock
VK_NUMLOCK	0x1D	Number lock
VK_ENTER	0x1E	Enter key
VK_SYSRQ	0x1F	System request
VK_F1	0x20	Function key F1
VK_F2	0x21	Function key F2
VK_F3	0x22	Function key F3
VK_F4	0x23	Function key F4

*(continued)*

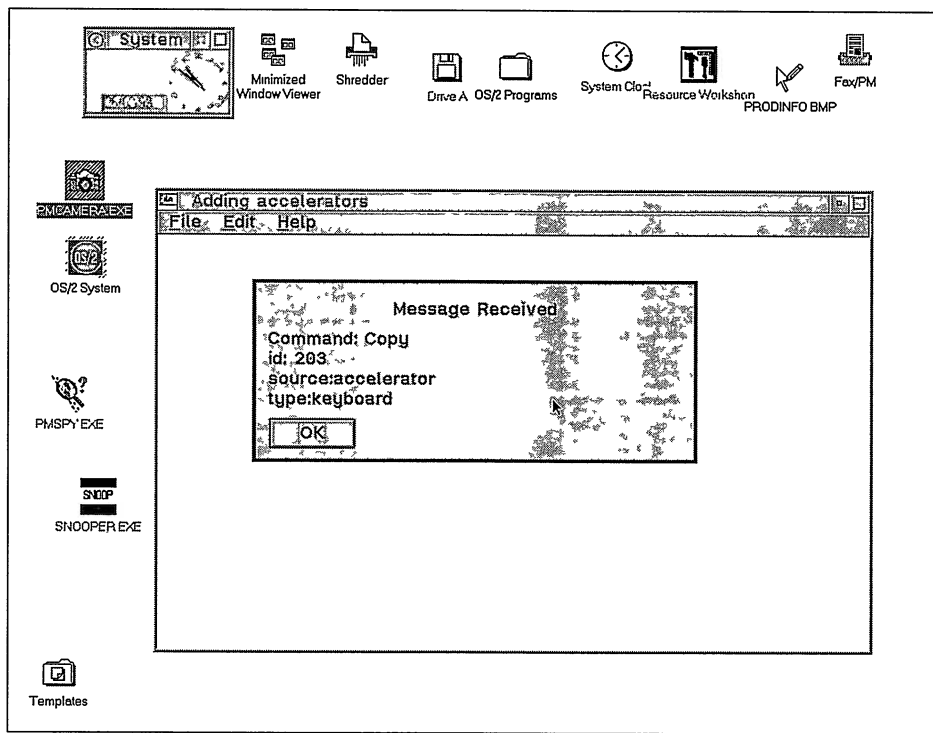
Table 6.13 (Continued)

Key	Value	Description
VK_F5	0x24	Function key F5
VK_F6	0x25	Function key F6
VK_F7	0x26	Function key F7
VK_F8	0x27	Function key F8
VK_F9	0x28	Function key F9
VK_F10	0x29	Function key F10
VK_F11	0x2A	Function key F11
VK_F12	0x2B	Function key F12
VK_F13	0x2C	Function key F13
VK_F14	0x2D	Function key F14
VK_F15	0x2E	Function key F15
VK_F16	0x2F	Function key F16
VK_F17	0x30	Function key F17
VK_F18	0x31	Function key F18
VK_F19	0x32	Function key F19
VK_F20	0x33	Function key F20
VK_F21	0x34	Function key F21
VK_F22	0x35	Function key F22
VK_F23	0x36	Function key F23
VK_F24	0x37	Function key F24

Associating an accelerator table with a window is a very simple operation. As with other resources like MENU and ICON, you only need to set the FCF\_ACCELTABLE flag (in addition to the appropriate ID) in order to make *WinCreateStdWindow()* perform the trick. The implementation is quite straightforward. In Listing 6.9 you can see the ACCEL application, similar to TWOMENUS. There are always two menu bars, but only one of them has an accelerator table. This further complication allows us to analyze the rules to follow for loading an accelerator table dynamically.

In general, this operation is performed with the *WinCreateStdWindow()* function. Accelerators are in fact a resource that can be hooked to a frame window or to the message queue of the system. Most often you will choose the first solution, leaving the second one for processing standard accelerators like those for the titlebar menu. The flag FCF\_STANDARD also includes FCF\_ACCELTABLE. When you create the main window—equipped with the first menu—then the accelerator table is also loaded automatically. By pressing the accelerator ALT+BACKSPACE, you will produce the effect shown in Figure 6.20.





**Figure 6.20** The message box will present information that allows you to identify the source of selection of the menu item.

The switch to the second menu causes a problem in the way accelerators work. Since this kind of resource is associated with the frame window and not with menus (although it appears to be more closely related to this second kind of window), pressing the key combination ALT+BACKSPACE will always cause the display of the message box shown in Figure 6.20. To avoid this *bug* you have to destroy the accelerator table when you load the second menu. When you get back to the first menu, you will have to reload the accelerator table from the resource file. To perform these operations, you can call `WinLoadAccelTable()`, `WinDestroyAccelTable()`, `WinQueryAccelTable()`, and `WinSetAccelTable()`, as you can see in Listing 6.9. An accelerator table can also be created dynamically during the application's execution by calling `WinCreateAccelTable()`. Except for some very special cases, it is better to create accelerators directly in the resource file—it's also more convenient.

---

## Run-Time Menus

Up to this point, the term *menu* has indicated both top-level menus in the menu bar and the drop-down menus that are displayed after the selection of a top-level menu.

By examining these two windows in detail, you will find there are substantial differences, both behavioral and functional. Top-level menus almost always imply a drop-down menu, sometimes called a *submenu*. The options of a submenu can be simple menu items or even other submenus that introduce other menu items.

Each top-level menu contains, among the information that characterizes it, the handle of the associated drop-down window. This handle is used to change the parent of the drop-down when the user selects through the mouse or the keyboard the area occupied by the menu bar. This action is followed by the display of the drop-down menu. On the basis of this, it is natural to think about creating a drop-down type window and assigning it any portion of the screen. The only operation that you need to complete is that of defining the parent and the owner window as the client window of the application, rather than the desktop window and the menu bar. The result is what appears in Figure 6.21, where a menu is created through *WinCreateWindow()*.

The class *WC\_MENU* does not have a predefined data structure. In order to be able to use *WinCreateWindow()* and the *WC\_MENU* class, you must define a data structure of your own to describe the general information regarding the menu and each menu item. Listing 6.10 refers to the application in Figure 6.21.

The creation of a menu window can take place only after compiling a dynamically allocated memory area with the appropriate information for defining each menu item. Once this operation is terminated, the returned handle is used in the program dynamically

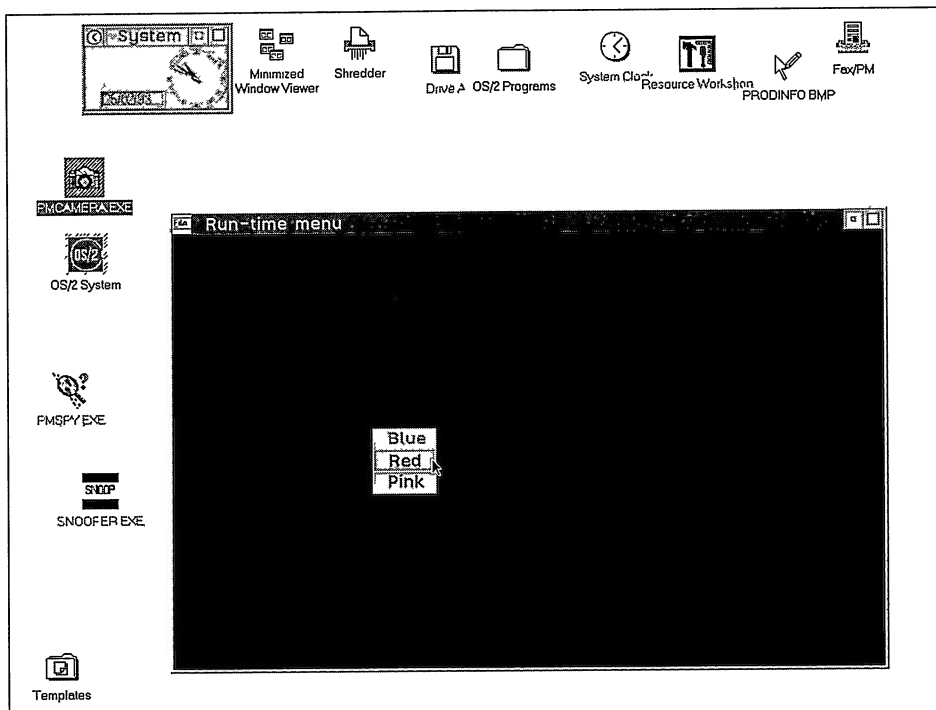


Figure 6.21 A drop-down menu positioned inside the client area of a window.

to display a drop-down menu each time the user clicks the left mouse button. The menu appears selected with the current color of the client window, due to the sending of the `MM_SELECTITEM` message. The choice of one of the three colors will change the background of the client window of the application.

---

## WPS Menus

At this point we can start to have fun, and give our PM application a look similar to those of WPS objects. This will keep us busy, but the outcome will be rewarding! The operating system does not furnish any program that fully complies with the style rules introduced by WPS. The system editor, `E.EXE`, is an old 16-bit application. The advanced editor, `EPM.EXE`, has many similarities to text-based products, with only a few interesting functions, like dragging the titlebar.

The first objective to reach is that of replacing the standard system menu with something more customized and closer to the application. The same menu must be able to appear in any position of the client area in response to the user's pressing the right mouse button (2) or the key combination `SHIFT+F10`. The following must be done to accomplish this:

- Clear the `FCF_SYSMENU` flag when creating the main window
- Create a menu template in the resource file containing all the options specific to the window/application
- Intercept the `WM_CREATE` message and take care of loading the menu template, and associate it with the window
- Intercept the `WM_CONTEXTMENU` message to display the *window context menu*
- Provide for the display of the window context menu in response to the `SHIFT+F10` accelerator
- Emphasize the client window (optional)

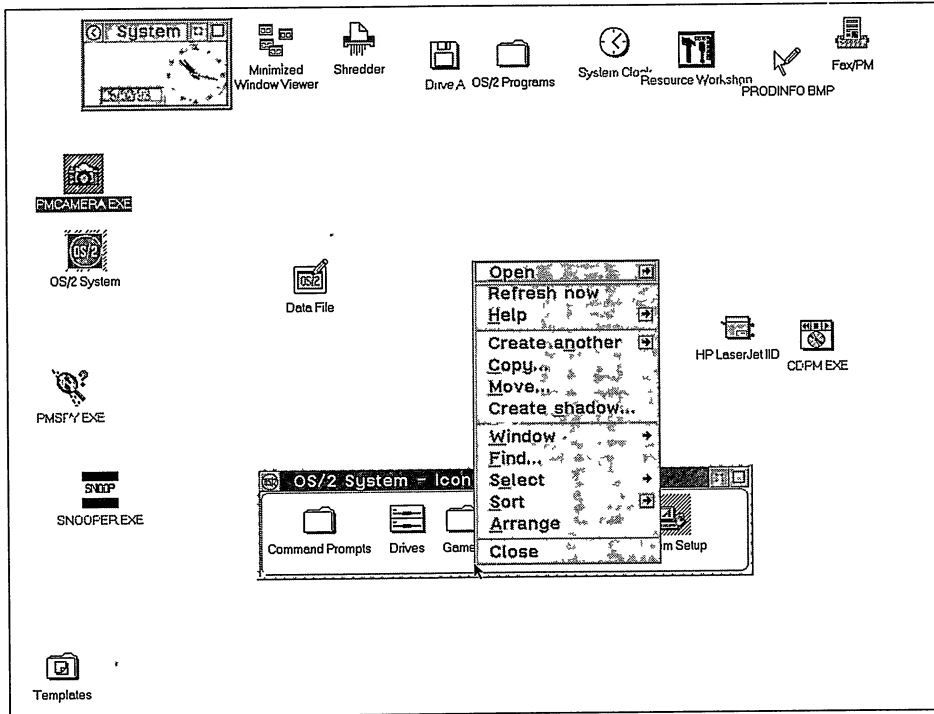


The system menu produced with `FCF_SYSMENU` contains only some of the information that appears in the much larger and comprehensive window context menu of the application. The solution implemented in Listing 6.11 allows you to prepare a menu template with an abundance of options directly in the resource file. These also include the options dealing with window management (Move, Size, and so on) as a secondary menu introduced by the Window item. This is a typical feature that can be found in all of WPS's folders (Figure 6.22).

The menu template also caters for the Help and Open options, in addition to many others that are specific to the application. The menu templates will present a numeric reference to a bitmap as its top-level menu. This is actually the icon that will appear to the left of the titlebar.

Without the `FCF_SYSMENU` flag, the window would not have the titlebar icon. This deficiency will not be apparent, though, because by using the `WM_CREATE` message you will actually create a new one. Menus are windows belonging to the predefined class





**Figure 6.22** The window context menu of the OS/2 System folder highlights the contents of the secondary Window menu.

WC\_MENU. As we will see in Chapter 7, the predefined classes have some unique features. In addition to the messages, styles, and attributes of the menu items, there will also be some styles that can be set at the moment the window is created (Table 6.14).

The syntax of the menu template does not provide any means for specifying the values given in Table 6.14. These are styles in the true sense of the term, and can be assigned to a window just like WS\_ styles. When creating a titlebar menu, set the MS\_TITLEBUTTON style in order to achieve the desired effect. First of all, let's load the

**Table 6.14** The Styles Available for Creating Windows of the WC\_MENU Class

<i>Style</i>	<i>Value</i>	<i>Description</i>
MS_ACTIONBAR	0x00000001L	Style of the menu bar.
MS_TITLEBUTTON	0x00000002L	Needed for the titlebar menu.
MS_VERTICALFLIP	0x00000004L	Allows a menu to be flipped vertically if the underlying space is insufficient.
MS_CONDITIONALCASCADE	0x00000040L	Produces a mini-pushbutton.

titlebar menu template from the RC file. *WinLoadMenu()* returns the handle of a window of the type `WC_MENU`, characterized by a bitmap as its top-level element. To add the new style, just retrieve the style packet that is already assigned, and reset it. The action can be conducted with the pair of functions *WinQueryWindowULong()* and *WinSetWindowULong()*:

```
...
u1Style = WinQueryWindowULong( hmenu, QWL_STYLE) ;
WinSetWindowULong( hmenu, QWL_STYLE, u1Style | MS_TITLEBUTTON) ;
...
```

This is not the only change that you need to make to the menu template. Conditional menus are introduced by a *mini-pushbutton* object, through the assignment of the `MS_CONDITIONALCASCADE` style, to an option described with the `SUBMENU` directive. The implementation of mini-pushbuttons as a class style is odd—it would have been simpler if it were assigned through the style of a single menu item! However, to set up a mini-pushbutton, we will develop a function, *SetConditionalMenu()*, that will take as parameters the handle of the menu and the ID of the menu item to change. Furthermore, it is necessary that the submenu always contain at least one menu item with the check-mark symbol. Pressing the mini-pushbutton will display the submenu, while a click on the text string corresponding to the `SUBMENU` will automatically execute the menu item with the check mark. Practical implementation relies on retrieving the style of the items that lead to the conditional submenus, and on the subsequent addition of `MS_CONDITIONALCASCADE`.

```
...
u1Style = WinQueryWindowULong( hmenu, QWL_STYLE) ;
WinSetWindowULong( hmenu, QWL_STYLE, u1Style | MS_CONDITIONALCASCADE) ;
...
```

The last operation to be performed is simple, though extremely important. You must make the frame window understand that the menu just built must be treated as a frame control. To do this you have to set the ID as the `FID_SYSMENU` value, so that the window gets equipped with a true titlebar menu:

```
...
WinSetWindowUShort( hmenu, QWS_ID, FID_SYSMENU) ;
...
```

In Figure 6.23 you can see the final result. To change an old and inadequate system menu into something more WPS-compliant, you really don't need to do much. The `CNTXT` application is dressed up with a menu bar and associated drop-down menus, according to the traditional development scheme.

---

## Interactions between the Menu Bar and the Client Window

The options that are typical of the system menu have all been collected in the second level drop-down menu introduced by `Window`. As this is a menu with the `FID_SYSMENUID`,

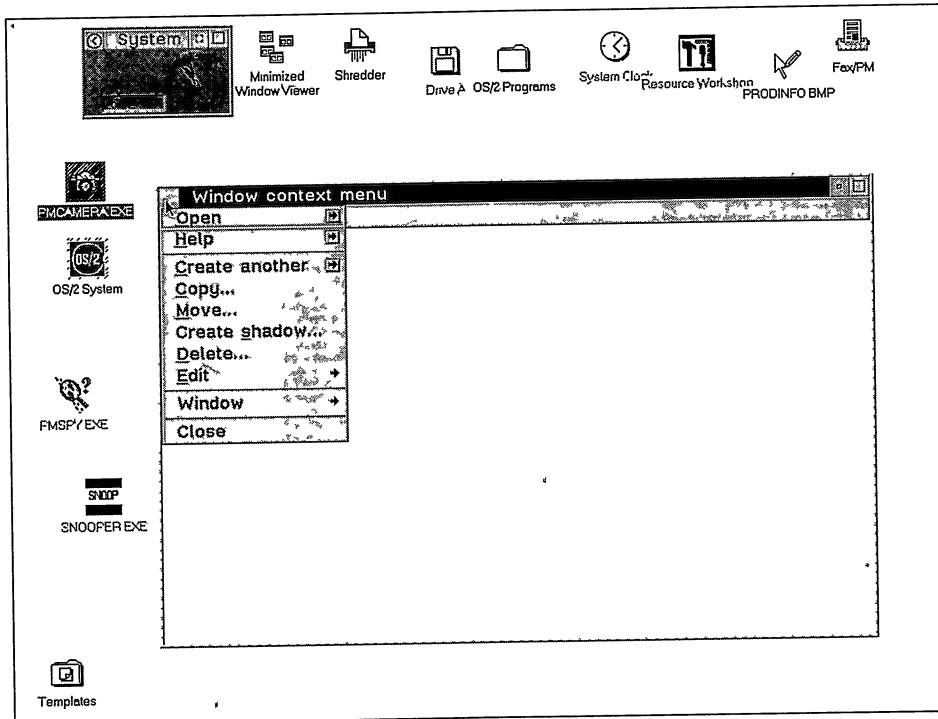


Figure 6.23 CNTXT has replaced the traditional system menu with a drop-down menu equivalent to its own window context menu.

PM itself will take care of dynamically enabling and disabling the various options, according to the window's current state. This is a great convenience. Furthermore, when writing the menu template, some options will have the `MIS_SYSCOMMAND` flag set to induce the `WM_SYSCOMMAND` message to be issued in place of the usual `WM_COMMAND` message. The selection of any menu item lacking the `MIS_SYSCOMMAND` flag will cause a `WM_COMMAND` to be sent to the frame window. However, the frame window will just forward the message to the client window. Consequently, any interaction with the titlebar menu can be detected and acted upon in the window procedure of the client window, just as if it were a menu item associated with the menu bar. Some options in the titlebar menu are identical to those available in the menu bar.

## Window Context Menu

With the changes described in the preceding paragraphs we have given our application a much better look, far more in tune with the style rules of WPS. Let's go on and implement the *window context menu*. The route we will follow, as always, is to get the

most with the least effort. The window context menu is identical to the titlebar menu, the only difference being the absence of the icon. We can therefore rely on the same menu template described in the resource file and load a second copy of it, always when the WM\_CREATE message is detected. A most important aspect to keep in mind is that of the value of the first parameter to pass to *WinLoadMenu()*. In the case of the titlebar menu we have indicated as the parent the handle of the frame window. For the window context menu, however, you must specify the handle of the desktop, *HWND\_DESKTOP*.

```
hpopup = WinLoadMenu( HWND_DESKTOP, NULLHANDLE, RS_TBMENU ) ;
```

Once you have the WC\_MENU class window, you can assign it an ID. However, any ID will not do: It must be, once again, *FID\_SYSMENU*.

```
WinSetWindowUShort( hpopup, QWS_ID, FID_SYSMENU ) ;
```

To get rid of the top-level (the icon), you have to query the menu with the message *MM\_QUERYITEM*. All information that describes it will be contained in a *MENUITEM* structure.

```
WinSendMsg( hpopup, MM_QUERYITEM,
            MPFROM2SHORT( FID_SYSMENU, FALSE ),
            MPFROMP( &mi ) ) ;
```

Herein you will find the handle of the associated drop-down menu, which is the window you want to display in response to the user's pressing the right mouse button or the SHIFT+F10 keyboard combination.

```
hpopup = mi.hwndSubMenu ;
```

The whole is completed by assigning mini-pushbuttons to the appropriate menu items. The identifier *hpopup* will initially represent the whole menu. Later, after assigning the contents of the *hwndSubMenu* member of the *MENUITEM* structure, it will identify the window context menu. All of this processing is placed inside the code dealing with the WM\_CREATE message, and the *hpopup* identifier is assigned static storage class in order to simplify displaying the window context menu. In practice, the application looks up the same menu template both for its titlebar menu as well as for its window context menu, thereby ensuring that the two objects are the same.

---

## Detecting WM\_CONTEXTMENU

The sequence that generates the message *WM\_CONTEXTMENU* has already been examined in Chapter 5, when studying the mouse. In *mp2* you will have the mouse position expressed in window coordinates, provided the message was actually generated with the mouse (TRUE in *mp1*). When intercepting *WM\_CONTEXTMENU* you must keep this distinction in mind, which forces us to query the position of the pointer on the screen with *WinQueryPointerPos()* if the source was the SHIFT+F10 keyboard combination.

```
#define INCL_WINPOINTERS
BOOL APIENTRY WinQueryPointerPos( HWND hwndDesktop, PPOINTL ppt1 ) ;
```

<i>Parameter</i>	<i>Description</i>
hwndDesktop	Handle of the desktop window or HWND_DESKTOP
pptl	Address of a POINTL structure
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

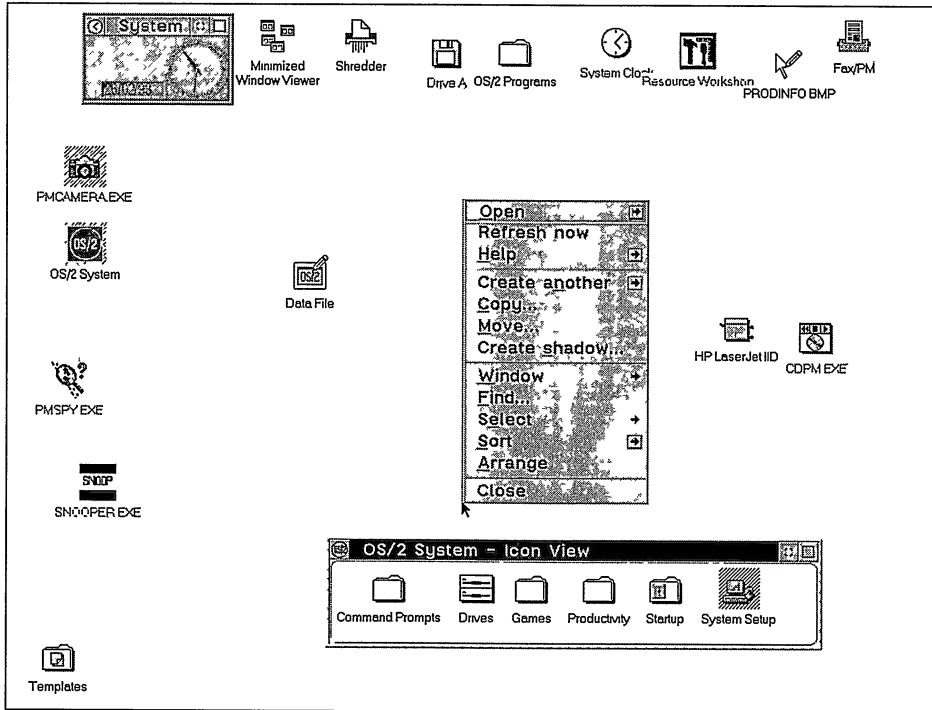
In the POINTL structure you will find the mouse's position on the screen, expressed in desktop units. This is an optimal situation for a subsequent call to *WinPopupMenu()*. If, on the other hand, the user has pressed the right mouse button, the information will be contained in mp2, but this time in window units. You have two possible solutions. Either you query the mouse's position again with *WinQueryPointerPos()*, or, in the case of a right mouse button click, retrieve the values from mp2 and convert them with *WinMapWindowPoints()*. The two alternatives are equivalent. The window context menu will always have the desktop as its parent, while its owner is one of the application's windows. The only thing left to do is to call *WinPopupMenu()* to display the window context menu.

```
#define INCL_WINWINDOWMGR
BOOL APIENTRY WinPopupMenu(  HWND hwndParent,
                             HWND hwndOwner,
                             HWND hwndMenu,
                             LONG x,
                             LONG y,
                             LONG idItem,
                             ULONG fs) ;
```

<i>Parameter</i>	<i>Description</i>
hwndParent	Handle of the parent window, generally HWND_DESKTOP
hwndOwner	Handle of the owner window, generally the <i>frame</i> window of the application
x	Lower left corner of the menu on the X axis
y	Lower right corner of the menu on the Y axis
idItem	ID of the item to be selected by default
fs	One or more PU_ flags
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The first two handles define the parent and the owner. For a window context menu, the best choices are HWND\_DESKTOP and the frame window, respectively. In fact, this menu can also appear outside the main window, as you can see in Figure 6.24.

By defining the frame window as the owner, you ensure that you will get the same results previously seen for the titlebar menu. The messages with the MIS\_SYSCOMMAND style will reach the frame window, while all others will be submitted to the client window. Furthermore, this solution also has the advantage of delegating to PM all handling of the menu items in the Window submenu.



**Figure 6.24** By pressing the SHIFT+F10 keys you can display an object's window context menu, even if the mouse pointer is completely outside of its window.

The position of the lower left corner of the menu must be expressed in units of the parent window. Since `HWND_DESKTOP` has been chosen as the parent window, the coordinates are expressed in screen units. The fifth parameter corresponds to the menu item you want to be selected by default. The value 0 will automatically select the first item. The syntax of `WinPopupMenu()` is completed with the flags listed in Table 6.15.

After intercepting the message `WM_CONTEXTMENU`, the interactions with the mouse buttons are terminated. For this reason, you never have to use the flag `PU_MOUSEBUTTONxDOWN`. Hence, the user will find in the window context menu functionality similar to that of the WPS objects. The selection of menu items can be performed both with the left mouse button (`PU_MOUSEBUTTON1`) as well as with the right button (`PU_MOUSEBUTTON2`), in addition to the keyboard (`PU_KEYBOARD`). The menu always appears where the mouse's hot spot is, unless it cannot fit in the available space. The flags `PU_HCONSTRAIN` and `PU_VCONSTRAIN` are very useful, while the flag `PU_POSITIONITEM` is not recommended for a window context menu. The `PU_SELECTITEM` flag is not useful, since WPS rules require that the first menu item always be selected. In practice, the set of flags that can be used is the following:

```
PU_NONE | PU_HCONSTRAIN | PU_VCONSTRAIN | PU_MOUSEBUTTON1 |
PU_MOUSEBUTTON2 | PU_KEYBOARD
```

**Table 6.15 The PU\_Flags Used for Handling the Window Context Menu**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
PU_NONE	0x0000	The button that displayed the menu does not interfere with its processing.
PU_POSITIONONITEM	0x0001	Positions the menu so that the <i>idItem</i> menu item appears exactly under the mouse pointer.
PU_HCONSTRAIN	0x0002	Changes the position of the menu on the X axis in order to ensure the whole object is visible.
PU_VCONSTRAIN	0x0004	Changes the position of the menu on the Y axis in order to ensure the whole object is visible.
PU_MOUSEBUTTON1DOWN	0x0008	The menu is initialized when the left mouse button is released.
PU_MOUSEBUTTON2DOWN	0x0010	The menu is initialized when the right mouse button is released.
PU_MOUSEBUTTON3DOWN	0x0018	The menu is initialized when the center mouse button is released.
PU_SELECTITEM	0x0020	Evaluates the <i>idItem</i> parameter as the ID of a menu item, causing it to be selected.
PU_MOUSEBUTTON1	0x0040	Enables the left mouse button to select the menu items.
PU_MOUSEBUTTON2	0x0080	Enables the right mouse button to select the menu items.
PU_MOUSEBUTTON3	0x0100	Enables the center mouse button to select the menu items.
PU_KEYBOARD	0x0200	Enables the keyboard to select the menu items.

The function *WinPopupMenu()* will return immediately and leave the menu displayed on the screen. In such a situation, the application will have lost the focus, now transferred to the menu. To make the window context menu disappear, you have choices other than selecting a menu item. Pressing the ESC key will terminate all operations of the *menu mode* kind (i.e., focus on a window of class WC\_MENU). The same holds true for clicking any of the mouse buttons in any place outside the menu. It is important to stress another peculiarity. The activation of the window context menu with SHIFT+F10 will cause the message WM\_INITMENU, and then WM\_MENUEND, to be received by the client's window procedure, although these two messages are addressed to the frame only, in response to a click on the mouse's right button. In Figure 6.25 you can see the CNTXT application which displays the titlebar menu; in Figure 6.26 you can see the identical window context menu.

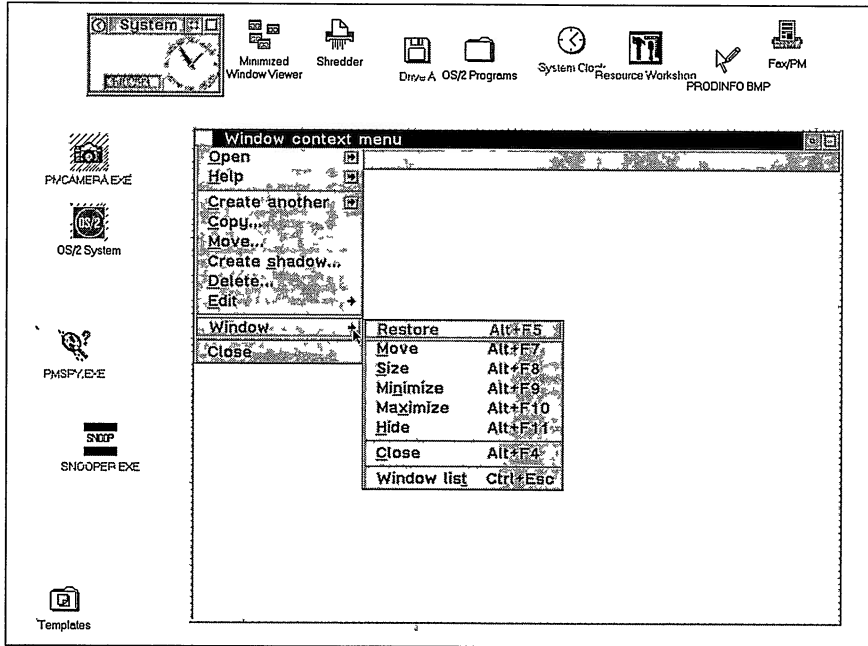


Figure 6.25 The titlebar menu of CNTXT.

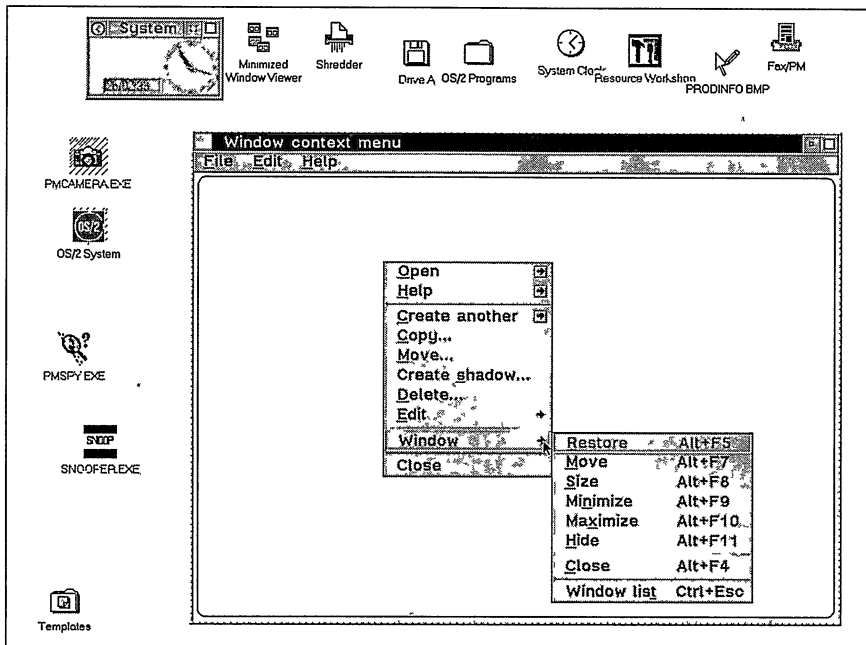


Figure 6.26 The window context menu of CNTXT.



## Emphasizing an Object

The last case to consider is how to emphasize the client window when the window context menu is displayed. Before describing the code required for this, we should discuss some considerations. In WPS only a few objects actually have this kind of menu. In addition to folders, the complete list includes the system clock (but not the color and font palettes, for instance), and a few other objects. Pressing the right mouse button in a container will cause an object to be emphasized by a rounded-angled frame. The same is not true for the clock. The style rules are somewhat vague here. The general case is that of emphasizing any application/object. Look at Figure 6.27. To what object does the menu refer? Probably not the desktop, because the rounded frame is missing; but it is also not intuitive to associate it with the clock.

The presence of an emphasizing border would have solved this problem. And that is what is going on in the CNTX example (Figure 6.28).

To draw the rectangle, we can call the *GpiBox()* function, which is both simple and flexible to use.

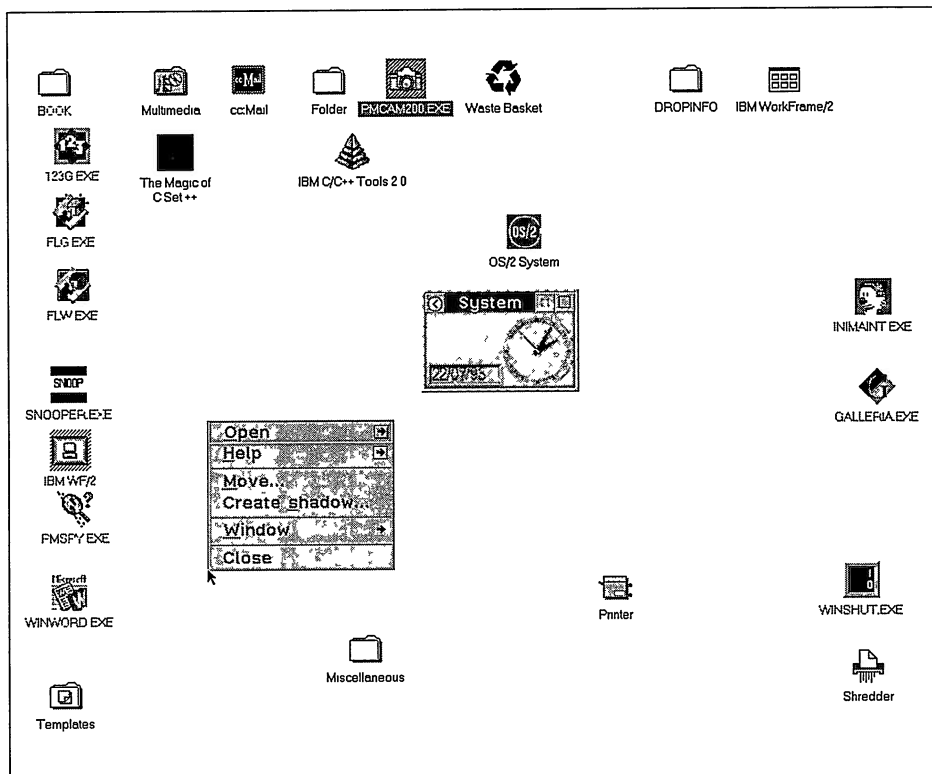
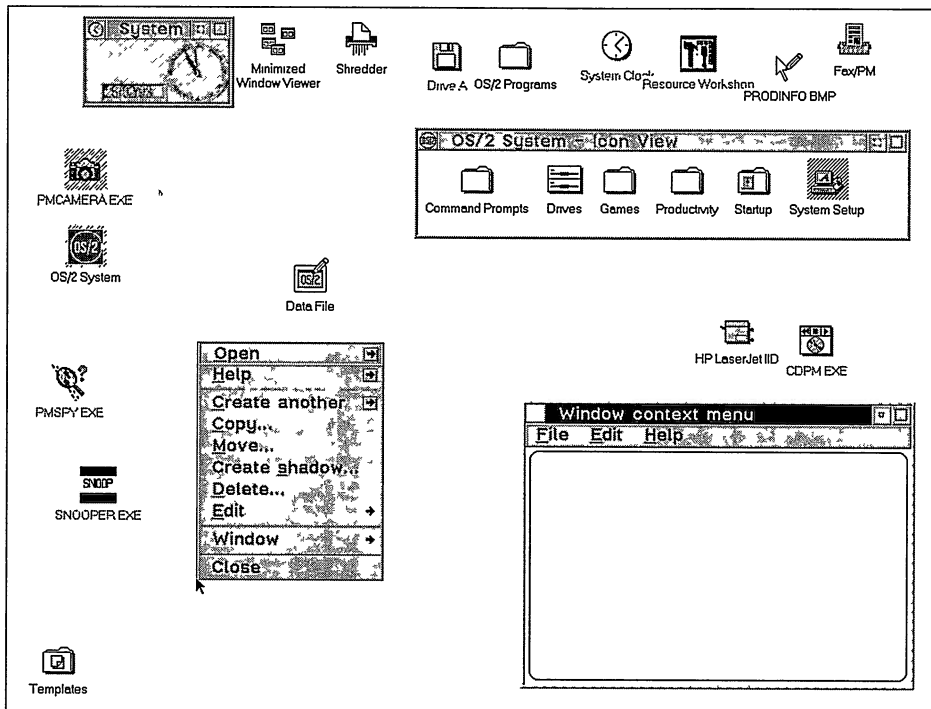


Figure 6.27 The window context menu of an object can also appear outside of the object.



**Figure 6.28** Even if the menu is completely outside, it is clear which object is its owner, thanks to the emphasizing border.

```
#define INCL_GPIPRIMITIVES
LONG APIENTRY GpiBox( HPS hps,
                    LONG lControl,
                    PPOINTL pptlPoint,
                    LONG lHRound,
                    LONG lVRound) ;
```

<i>Parameter</i>	<i>Description</i>
hps	Handle to the presentation space
lControl	Define with the prefix DRO_
pptlPoint	Angle opposite to the current position in the presentation space
lHRound	Quantity used to determine the approximation of the angle on the X axis
lVRound	Quantity used to determine the approximation of the angle on the Y axis
<i>Return Value</i>	<i>Description</i>
LONG	GPI_xxx define

**Table 6.16** The Defines for the Second Parameter of *GpiBox()*

<i>Flag</i>	<i>Value</i>	<i>Description</i>
DRO_FILL	1L	The rectangle is filled with the current color.
DRO_OUTLINE	2L	Only the outline is drawn.
DRO_OUTLINEFILL	3L	The outline is drawn and the rectangle is filled.

The handle to the presentation space is obtained through *WinGetPS()*, and subsequently released with *WinReleasePS()* directly in the code of the *WM\_CONTEXTMENU* message. In this case, there aren't any *painting* problems, since any action carried out on the menu items, on other portions of the window, or on another window, will always produce just one effect: issuing message *WM\_SETFOCUS*. The emphasizing border is cleared in the code fragment associated with the *WM\_SETFOCUS* message (since it is not possible to use either *WM\_INITMENU* or *WM\_MENUEND*, because these messages will not always reach the client's window procedure). In Table 6.16 you can see the defines used by the *GpiBox()* function.

At this point you only need to give a final look at the whole of *CNTXT*'s code (Listing 6.11), and always remember to build window/objects equipped with a window context menu.



# The Predefined Window Classes

In PM there are as many as 15 predefined window classes, all described in PMWIN.H and PMSTDDL.G where they are identified by the WC\_ prefix (Table 7.1). The actual design of the predefined window classes was done by IBM's engineers following the traditional plan that we have seen in the previous chapters. The registrations and the window procedures of the predefined window classes are all to be found inside system DLLs, which are always loaded when OS/2 boots. This gives the predefined window classes a total and permanent visibility and accessibility from any executable or DLL module subsequently loaded.

**Table 7.1 The 15 Predefined Window Classes in OS/2 2.1**

<i>Class</i>	<i>Value</i>
WC_BUTTON	((PSZ)0xffff0003L)
WC_COMBOBOX	((PSZ)0xffff0002L)
WC_CONTAINER	((PSZ)0xffff0025L)
WC_ENTRYFIELD	((PSZ)0xffff0006L)
WC_FRAME	((PSZ)0xffff0001L)
WC_LISTBOX	((PSZ)0xffff0007L)
WC_MENU	((PSZ)0xffff0004L)
WC_MLE	((PSZ)0xffff000aL)
WC_NOTEBOOK	((PSZ)0xffff0028L)
WC_SCROLLBAR	((PSZ)0xffff0008L)
WC_SLIDER	((PSZ)0xffff0026L)
WC_SPINBUTTON	((PSZ)0xffff0020L)
WC_STATIC	((PSZ)0xffff0005L)
WC_TITLEBAR	((PSZ)0xffff0009L)
WC_VALUESET	((PSZ)0xffff0027L)

The main advantage of having a significant and assorted set of predefined classes is obvious. In the first place, many of the typical functionalities of an application—like handling keyboard input—can easily be delegated to these windows. It is quite unlikely that an application will be identified wholly with one of the predefined window classes. Most often, this kind of window shows up as a child window of some other window. For this reason the term *control* is often used to stress the role of *control elements* that these windows play within a PM program.

The application designer does not need to know anything about the internal logic and structure of the predefined class's window procedures, but will need to know all mechanisms and rules for using them:

- Appropriate messages
- Notification codes
- Window styles
- Data structures

Each of these predefined classes is characterized by a set of messages of its own, in addition to the standard WM\_ messages. These messages, which are introduced by a class-specific prefix, are the main tool by which the software designer interacts with a control. The notification codes are sent from the control's window procedure to the owner's window procedure in order to tell it how the received message was actually processed. The whole picture is then completed by a group of styles used to customize the window's look, and, in some cases, by data structures that set some window behaviors (Table 7.2).

**Table 7.2 The 15 Window Classes and Their Associated Data Structures**

<i>Class</i>	<i>Data Structure</i>
WC_BUTTON	btncd
WC_COMBOBOX	Absent
WC_CONTAINER	multiple
WC_ENTRYFIELD	efd
WC_FRAME	fcdata
WC_LISTBOX	Absent
WC_MENU	Absent
WC_MLE	Absent
WC_NOTEBOOK	Absent
WC_SCROLLBAR	sbcd
WC_SLIDER	sldcdata
WC_SPINBUTTON	Absent
WC_STATIC	Absent
WC_TITLEBAR	Absent
WC_VALUESET	vscdata

Before getting into the nuts and bolts of each predefined window class, it is important to understand the driving philosophy behind these windows.

---

## The Message Flow

One of the concepts that has been stressed in the first chapters, is message flow. A window procedure will receive all messages pertaining to its class, one at a time.

Imagine a message as an airplane and the window procedure as the landing strip. No matter how heavy the air traffic might be, only one airplane at a time can utilize the landing strip. It is fundamental that any type of airplane (message) can find in the landing strip (the window procedure) all elements that it needs (adequate length, distance signs, light guides and so on). The messages will run through the window procedure and in some cases will be intercepted, or they will flow directly to `WinDefWindowProc()`; often, during their processing, they cause yet other messages to be issued. This logic is valid even for the windows belonging to the predefined classes. Though there is a fundamental difference with respect to the window procedure of a generic client window, it is impossible for the designer to “trace” in any way the flow of messages passing by (the airplanes do not land on some landing strips of our airport).

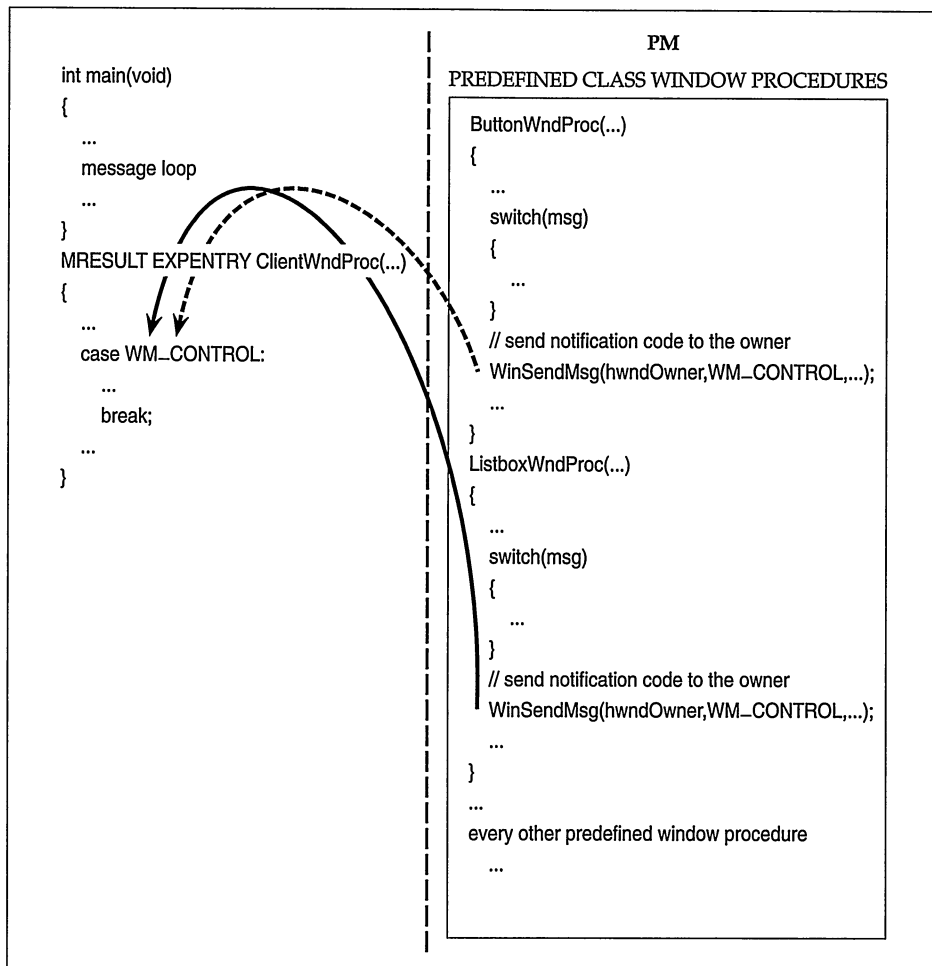
Let’s examine how the same action can have different effects when performed on a window of the one class or the other. A double-click of the left mouse button on the titlebar (a window of class `WC_TITLEBAR`) will cause the window to be maximized or restored on the screen, according to its current state. A double-click on the client window will not produce any effect. In the first case, this means that the window procedure of the class `WC_TITLEBAR` contains a case condition for `WM_BUTTON1DBLCLK` where the described behavior is implemented. Actually, this is only a hypothesis that is based on the experience and on the study of the functional model of PM; it is not based on any direct knowledge of the internal workings. Furthermore, we know that the mouse is a kind of independent input device that will invariably report its own position to the window over which the pointer is located at that very moment (unless, naturally, the mouse is captured). The double-click performed on the titlebar is an action that pertains exclusively to the class `WC_TITLEBAR` and its window procedure. The final result of this operation is the maximizing/restoring of the window, which affects the frame window, the client window, and all other window components that show up on the screen. How does this interaction among windows of different classes take place? The explanation is to be found in the criteria that were followed when the predefined class’s window procedure was designed. Once a message has been received, this category of window procedure will almost invariably issue many other messages to the child windows and to the owned windows. This explains why the action is so well coordinated between the titlebar, the frame, and all controls. However, this is not the only reason. Once the processing phase is over, each window procedure of any predefined window will notify its owner as to what action took place. That’s how the frame window is able to “understand” that it must maximize or restore

itself on the screen. Figure 7.1 illustrates the relationship that exists between a window procedure of a window belonging to a predefined class and that of the owner window.

The analysis of the nature of the menu windows in Chapter 6 gave you some clues about this kind of relationship, especially the interactions that take place between the drop-down menus and the menu bar.

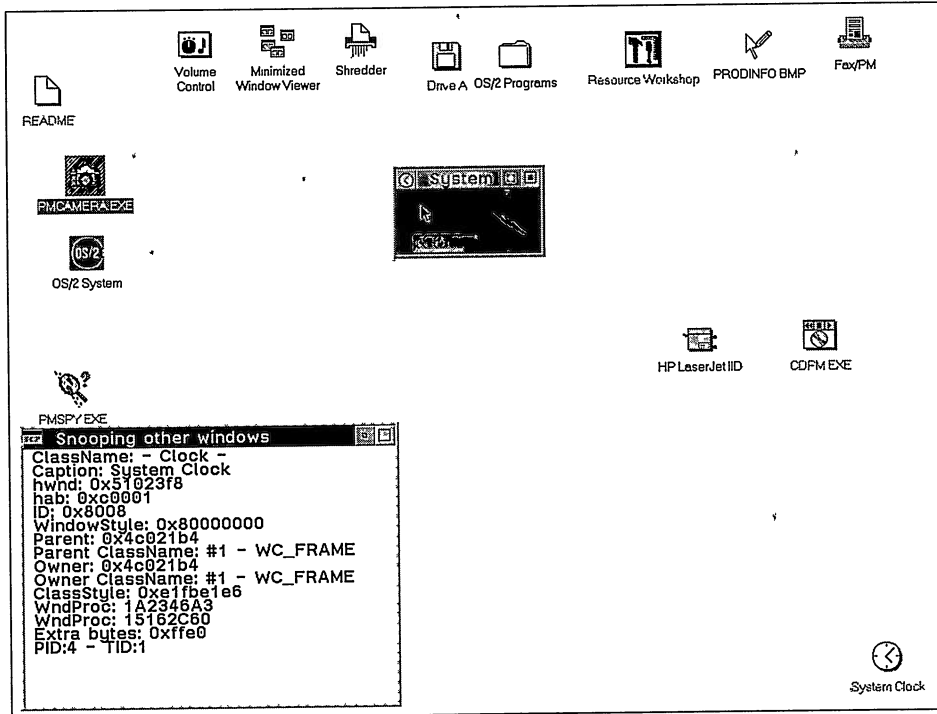
## Peeping through Windows

Before examining the windows belonging to the predefined classes, let's study the properties of the data produced by PM when creating a window. For this purpose,



**Figure 7.1** Scheme of interactions among windows related through an ownership relationship.





**Figure 7.2** In the client window of SNOOPER you can see lots of information regarding the structural elements of a window.

you can try the SNOOPER application, which is a simple PM utility that allows you to discover some information that might be interesting for anyone developing software for this environment. Listing 7.1 presents the source code of SNOOPER, which is shown in action in Figure 7.2.

SNOOPER does not look very sophisticated. Its output is just a series of text strings written directly in the client window of the application, thanks to *GpiCharStringAt()*. The functions used in the code of SNOOPER retrieve the information displayed in the client window; these functions are *WinQueryClassName()*, *WinQueryClassInfo()*, *WinQueryWindowUShort()*, *WinQueryWindowULong()*, *WinQueryWindow()*, and others.

The logic that governs SNOOPER is based on capturing the mouse input, an operation that is implemented by intercepting each single mouse button click. A press on the right mouse button will free the mouse and restore the standard operating rules. Once the mouse has been captured (a click on the left button), any operation performed with it will be addressed to the client window of SNOOPER. By exploiting the design described in Listing 5.2 for capturing the mouse, you then proceed by identifying the window underlying the mouse pointer at any given moment, via *WinWindowFromPoint()*. Actually, in OS/2 2.1, there is always some window under the mouse's hot spot. The system will always find at least one of these windows, in this order:

- The desktop of OS/2 PM (HWND\_DESKTOP)
- The frame window of WPS (child of HWND\_DESKTOP)
- A window of class WC\_CONTAINER, client of the preceding frame window

Of these three windows, two are always invisible and inaccessible, but the information regarding them can always be retrieved by climbing up the family tree of the windows.

Apart from the window identified by SNOOPER, all information retrieved through the above-mentioned functions is stored in a structure named SNOOP, which is defined in the application's header file together with an adequate pointer.

```
...
typedef struct _SNOOP
{
    CHAR szClass[ 34] ;
    CHAR szText[ 34] ;
    HAB hab ;
    CLASSINFO clsi ;
    PFNWP pfnWndProc ;
    HWND hwndOwner ;
    CHAR szOwnerClass[ 34] ;
    HWND hwndParent ;
    CHAR szParentClass[ 34] ;
    ULONG ulStyles ;
    USHORT usID ;
    HWND hwnd ;
    PID pid ;
    TID tid ;
} SNOOP ;

typedef SNOOP * PSNOOP ;
...
```

The members of the SNOOP structure are not difficult to understand, except perhaps for CLASSINFO, a structure which is specific to the *WinQueryClassInfo()* function, and that is described in PMWIN.H.

```
typedef struct _CLASSINFO
{
    // clsi
    ULONG flClassStyle ;
    PFNWP pfnWindowProc ;
    ULONG cbWindowData ;
} CLASSINFO ;
```

CLASSINFO contains the members *flClassStyle*, which describe the styles of the class ( a collection of CS\_ items specified during registration phase), *pfnWindowProc* to store the address of the class's window procedure, and *cbWindowData*, which holds the number of possible *window words* of the class. Among other pieces of information, you also have the *Process Identification Number* (PID), the *Thread Identification Number*

(TID), and the address of the window procedure, as it is stored in the window's reserved memory area. This last piece of information is actually redundant, but it does play a very important role because it allows you to determine whether or not this is a subclassed window.

The first information retrieved as soon as you get the handle of the window underlying the mouse's pointer is the name of the class to which the window belongs; this information is retrieved only after saving the window's handle in the member `snoop.hwnd`.

```
...
snoop.hwnd = hwndSnoop ;
lLen = (LONG)WinQueryClassName( hwndSnoop, sizeof snoop.szClass,
                               snoop.szClass) ;
...
```

The *WinQueryClassName()* function takes on the following syntax:

```
#define INCL_WINWINDOWMGR
LONG APIENTRY WinQueryClassName( HWND hwnd,
                                LONG cchMax,
                                PCH pch) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of a window
cchMax	Size of the character array that will contain the name of the class
pch	Array of characters containing the name of the class
<i>Return Value</i>	<i>Description</i>
LONG	Number of characters in the name of the class

The first parameter identifies a window belonging to the class the name of which you want to discover. It is not possible to find the name of a class without having available a window of that class.

Once you have the name of the class, you can easily access the information passed to PM at registration time with *WinRegisterClass()*. The function *WinQueryClassInfo()* returns the data in a `CLASSINFO` structure:

```
#define INCL_WINWINDOWMGR
BOOL APIENTRY WinQueryClassInfo( HAB hab,
                                 PSZ pszClassName,
                                 PCLASSINFO pClassInfo) ;
```

<i>Parameter</i>	<i>Description</i>
hab	Handle to the application's anchor block
pszClassName	Name of a class registered in PM
pClassInfo	Address of a <code>CLASSINFO</code> structure
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The first parameter is the handle of the anchor block of the application that registered the class. The second parameter is the name of the class previously retrieved with *WinQueryClassName()*, and the last parameter is the address of a *CLASSINFO* structure.

```
...
WinQueryClassInfo( HAB( hwndSnoop), snoop.szClass, &snoop.clsInfo );
...
```

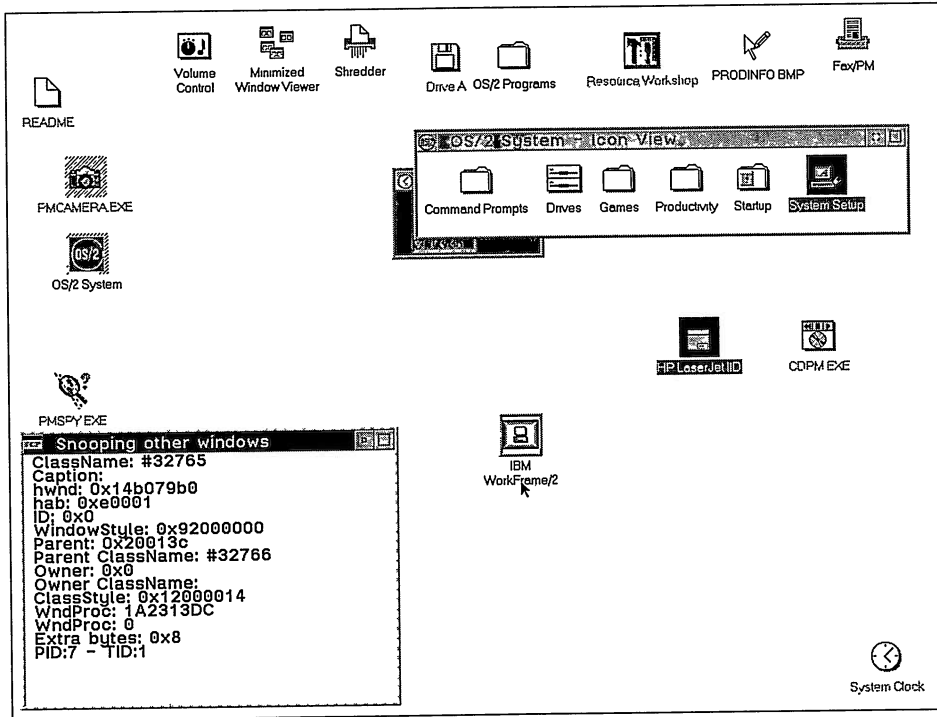
In Figure 7.2 you can see that the SNOOPER is not limited to indicating only the number of the class prefixed by the symbol #, but it will also provide its name like a *WC\_define*. The *STRINGTABLE* area of *SNOOPER.RC* accommodates the strings corresponding to the names of the predefined classes that have the same ID in *PMWIN\_H*:

```
...
STRINGTABLE
{
    1, " - WC_FRAME"
    2, " - WC_COMBOBOX"
    3, " - WC_BUTTON"
    4, " - WC_MENU"
    5, " - WC_STATIC"
    6, " - WC_ENTRYFIELD"
    7, " - WC_LISTBOX"
    8, " - WC_SCROLLBAR"
    9, " - WC_TITLEBAR"
    10, " - WC_MLE"
    32, " - WC_SPINBUTTON"
    38, " - WC_SLIDER"
    39, " - WC_VALUESET"
    37, " - WC_CONTAINER"
    40, " - WC_NOTEBOOK"
    100, " - undocumented"
    ...
}
...
```

The code fragment that loads the text string corresponding to the class is the following one:

```
...
if( !len < 4 && *snoop.szClass == '#' )
{
    int iClass ;

    if( ( iClass = atoi( snoop.szClass + 1) ) >> 0x28)
        iClass = 100 ;
    WinLoadString( HAB( hwnd), NULLHANDLE,
                  iClass, sizeof( szBuffer), szBuffer ) ;
    strcat( snoop.szClass, szBuffer ) ;
}
...
```



**Figure 7.3** The mouse is positioned over the text label under an icon: Its class is undocumented.

`WinLoadString()` is controlled by an `if` statement that checks whether the name of the class is shorter than four characters, not including the `#` symbol at the beginning. This preliminary test saves time by not searching for nonexistent defines when the mouse points to, for instance, a client window registered with a `#` at the beginning. The condition of the class name's length can be somewhat maddening. In PM, in addition to the 15 predefined classes, there are many more undocumented classes. Their names always start with a `#`, followed by a five digit number. SNOOPER will resolve the problem simply by displaying the number of the class, followed by the text "undocumented." Figure 7.3 shows information of an undocumented window class.

Once this first obstacle has been overcome, you can go ahead and store the remaining information in the SNOOP structure. First comes the handle of the *anchor block*, then the handle of the parent window, the class of the parent window, and many others.

When you have terminated all this data retrieval activity, you force the issuing of the `WM_PAINT` message by invalidating the whole client window. In the code dealing with `WM_PAINT` you prepare for displaying the text strings that constitutes the program's output. The text appears in the client area thanks to `GpiCharStringAt()`:

```

...
pt.x = 10L ;
pt.y = rc.yTop - ITEMHEIGHT ;
lLen = sprintf( szString, "ClassName: %s", snoop.szClass) ;
GpiCharStringAt( hps, &pt, lLen, szString) ;
...

```

In Chapter 13 you will see a revisited version of SNOOPER, one that is closer to the development model of WPS and has some very interesting features.

---

## The Structure of Windows

For a developer, an application like SNOOPER is vital. The information gathered through it can be a source of inspiration. Spying on other PM windows gives you a complete picture of the diverse ways in which you can exploit styles, flags, and options. Also, it is important to fully understand the relationship existing among the various windows.

Figure 7.3 shows SNOOPER displaying information regarding the #32765 of PM application. The next to last piece of information presented in the client windows indicates in hexadecimal notation the number of window words declared when the class was registered. Each window of the #32765 class has eight extra bytes in addition to the standard amount. It is not possible to know anything about their nature and contents. The only conclusion that you might draw is that these bytes might accommodate two LONG values... and that isn't that much of a discovery! It isn't even possible to be sure about the order in which the data items are stored in the window words and even less about their contents. Exploring other windows belonging to the predefined window classes, you can gather the results that are summarized in Table 7.3.

All of the predefined window classes have a number of window words which by far exceed the theoretical limit of 4 bytes given in Chapter 4. This discovery does not contradict the general rule. The predefined classes need to store lots more information because of the universal role they play. (Remember that on the landing strip of your airport there can be many different planes passing by—many messages sent to different windows belonging to the same class.) For a generic client window, however, it is wise not to exceed the limit. At the present stage, PMWIN.DLL—the DLL containing all API functions with the *Win* prefix—is still a 16-bit component, as you will be able to demonstrate with SNOOPWPS in Chapter 13. However, the linear addressing scheme that is typical of the 32-bit world and the internal functioning logic of PMWIN.DLL is still anchored to the old model of 64KB segments. Using window words is an optimal strategy for writing PM applications. Until PMWIN.DLL is converted to 32 bits, it is better to be prudent and avoid exhausting system resources. Eventually, when a 32-bit version of PMWIN.DLL does exist, however, applications will perform window

management much faster, and the software designer will have more horsepower to exploit and have fun with.

---

## The Predefined Window Classes and the Window Words

The functions *WinQueryWindowUShort()* and *WinQueryWindowULong()* and the corresponding functions for storing values all take into account the flags *QWS\_USER* and *QWL\_USER*. These two defines identify, respectively, a two-byte and a four-byte area inside the window words, which are available to the designer. The values of the window words column in Table 7.3 also include the space identified by *QWS\_USER* and *QWL\_USER*.

The strategy adopted by the designers of OS/2 PM was in this case a real reward for developers. The definition of window words is an operation that is allowed only when a class is being registered, and once that has already been performed for the predefined classes, the availability of four free bytes constitutes an opportunity for storing program-specific data even inside a predefined window, an entity not wholly under the designer's control. Take note of this: It will be very useful in the next few examples, and in developing complex applications.

**Table 7.3 The 15 Predefined Classes in PM and the Amount of Allocated Window Words**

<i>Class</i>	<i>Window Words</i>
WC_BUTTON	0x001a
WC_COMBOBOX	0x005a
WC_CONTAINER	0x000c
WC_ENTRYFIELD	0x0034
WC_FRAME	0x0060
WC_LISTBOX	0x0042
WC_MENU	0x0022
WC_MLE	0x0008
WC_NOTEBOOK	0x0008
WC_SCROLLBAR	0x0024
WC_SLIDER	0x0008
WC_SPINBUTTON	0x0008
WC_STATIC	0x000c
WC_TITLEBAR	0x000c
WC_VALUESET	0x0008

---

**Table 7.4 List of Some Undocumented Window Classes in OS/2 2.1**

<i>Class</i>	<i>Style</i>	<i>Description</i>
#32766	0xa0000000	The Desktop of PM.
#32765	0x92000000	Window containing a minimized running application's title.
#50	0x90000000	The Criteria window of the Include page of the Settings notebook (lower part).
#52	0x90000000	The Criteria of the Include page of the Setting notebook (upper part).
#54	0x80000000	Internal portion of a window belonging to the class WC_NOTEBOOK.
#55	0x80000000	Horizontal arrows for turning pages in a window belonging to the class WC_NOTEBOOK.
#57	0x80000000	Vertical arrows in a window belonging to the class WC_SPINBUTTON.

---

## Undocumented Classes

PM also has some undocumented classes. One of these, perhaps the most important one, is the #32766. The desktop—the ancestor of all PM windows—belongs to this class. In Table 7.4 you can see the names of some undocumented classes that you might be able to detect by using SNOOPER and other utilities. You should never try to use these classes, because there is little information about them, and they do not add much to the rich and powerful PM API.

You should not be concerned if there are undocumented elements in the API, because the missing information is not that important. Concentrate on the predefined classes, and you will see that the tools available to the designer are truly remarkable.

---

## How to Create a Window of Class WC\_

The “standard” windows created with *WinCreateStdWindow()* are equipped with various controls, like other windows belonging to predefined classes. The entire process is transparent to the user. When you need to explicitly create a listbox, a combobox, or an entryfield, then you need to call *WinCreateWindow()*.

Controls (with this term we usually exclude the classes WC\_FRAME, WC\_TITLEBAR, and WC\_MENU) seldom take the role of an application's main window. More often, they are accessory elements that might support the creation of a complex window equipped with several tools for interacting with the user, or they help in displaying information



retrieved or produced by the application's execution. The syntax of *WinCreateWindow()* has already been described in Chapter 4. Let's take a look at it again:

```

hwnd = WinCreateWindow(  hwndParent,
                          szClassName,
                          pszTitle,
                          ulStyles,
                          x, y,
                          cx, cy,
                          hwndOwner, hwndInsertBehind,
                          id,
                          pCtldata, pPresParam) ;

```

The return value is the handle of the new window, or `NULLHANDLE` in case of error. The first parameter is the handle of the parent window. Since the controls are almost never *top-level* windows, the pixels must be provided by some other window instead of `HWND_DESKTOP`. Often it will be directly an application's client window that takes the part of a control's parent.

The choice of a new window can range over any of the fifteen possible alternatives. The title, the third parameter, will be used in a broader sense. In all previous examples of the `WC_FRAME` class, the title was considered as the piece of text inserted in the titlebar. However, its real meaning depends on the class to which the window being created belongs. For the classes `WC_SCROLLBAR`, `WC_ENTRYFIELD`, `WC_MLE`, `WC_VALUESET`, and `WC_COMBOBOX`, you need to specify `NULL` in place of this parameter almost all the time. It does not make sense to contemplate a *scrollbar* with a title. It is very rare to assign a title to a window of the class `WC_LISTBOX`; on the other hand it is common for `WC_BUTTON` and `WC_STATIC`. The controls introduced by version 2.x are rather complex windows, that in some cases (`WC_CONTAINER` and `WC_NOTEBOOK`) require a preliminary preparation that is tedious and complex, and that eventually brings the title being assigned.

The position and the size of a window is always expressed in units of its parent, starting from the lower left-hand corner. A control that always preserves the same position and size, with respect to its parent, is usual. Less common is the case where a control can move about in the space it has available. In this case, they are rather called windows.

As far as the owner window's handle is concerned, the possibilities are even more complex. By using `SNOOPER` to examine some of the predefined windows, you might discover that the menu bar, the titlebar menu, and the sizing icons all have as their owner the application's frame window. Instead, the client window, created with *WinCreateStdWindow()*, does not have any owner, although it can acquire one by calling *WinSetOwner()*.

It is impossible to define a proper set of rules to which you can refer in defining the owner window of a control. Often the parent window plays double-duty as owner, even though this is not an absolute rule (just think of drop-down menus where the two roles are distinct).

The next handle, `hwndInsertBehind`, defines the window's on-screen position. In addition to specifying a handle, other solutions that are frequently employed are `HWND_TOP` and `HWND_BOTTOM`. In the first case, the window being constructed appears above all other windows at the same hierarchical level; in the second case it will be the last one in the list. The next parameter of `WinCreateWindow()` corresponds to the ID assigned to the windows. We have never dealt with the window ID before, because it is not possible to specify it directly with `WinCreateStdWindow()`. However, this information is necessary for all controls, as it gives to its parent another tool for accessing it. From the parent you can get to a child window by calling `WinWindowFromID()` and providing the correct ID. It is therefore vital to assign a unique ID to each control.

The twelfth is a pointer to an area allocated for accommodating data specific to the window's class. Some of PM's predefined classes have in `PMWIN.H` and `PMSTDDL.G.H` related data structures. This is the case, for instance, of `WC_FRAME`, but not of `WC_MENU` as you can see in Listing 6.10. The syntax of `WinCreateWindow()` is completed by the address of a memory area containing the presentation parameters. By this expression we indicate all information that lets you vary the color and/or the default font of a control. First introduced with OS/2 PM 1.2, the presentation parameters play a central role when building a window consisting of several controls.

## *When to Create a Window of a Predefined Class*

Any place in the code of a window procedure is good for placing a call to `WinCreateWindow()` to create a control. If the parent is a client—the most frequent situation—the best place is in the code fragment dealing with `WM_CREATE`. There you have available all information regarding the parent, which often is also the owner. Let's now examine the single characteristics of the various predefined classes, and study how they are best put to use.

---

## The Class `WC_BUTTON`

To this class belong several windows known as *buttons* that differ in shape and look. The differences are a consequence of setting certain styles when a window is created (Table 7.5). When building a window with `WinCreateStdWindow()` there is no representative of this class, even if their presence in commercial applications is continually growing (toolbars and similar objects often contain many buttons).

A window of the class `WC_BUTTON` will always appear in one of two possible states: pressed or not pressed. These two conditions are displayed on the screen in different ways related by the styles that characterize a button. There are, in fact, several buttons that have a three-dimensional, circular, or square look, and offer the designer different graphical and functional solutions. The shape of the buttons depends, in the first place, on the action of certain style flags when the window is created and on the state in which it is created.

**Table 7.5 The Styles of the Class WC\_BUTTON**

<i>Style</i>	<i>Value</i>	<i>Description</i>
BS_PUSHBUTTON	0L	Defines a <i>pushbutton</i> , i.e. a button that once selected issues the message WM_COMMAND to its owner window.
BS_CHECKBOX	1L	Looks like a small square with some text to its right; when it is selected it issues the message WM_CONTROL to its owner window, which is responsible for changing the button's look on the screen.
BS_AUTOCHECKBOX	2L	It is similar to the style BS_CHECKBOX as far as the <i>control's</i> look is concerned and the issuing of the WM_CONTROL message to its owner window; automatically displays an X in the square when it is selected.
BS_RADIOBUTTON	3L	Creates a small circle with some text to its right; generally <i>radiobuttons</i> are used in one group; when selected it posts the message WM_CONTROL to the owner window, which is responsible for managing all other buttons in the group.
BS_AUTORADIOBUTTON	4L	Identical to BS_RADIOBUTTON with the only difference that the button is automatically highlighted and, if selected, removes the selection from any other <i>autoradio</i> button that belongs to the same group.
BS_3STATE	5L	Identical to BS_AUTOCHECKBOX as far as the <i>control's</i> look is concerned and the issuing of the WM_CONTROL message to its owner window; automatically displays an X when it is selected, but the selection also includes the empty square and a square filled with a gray pattern.
BS_AUTO3STATE	6L	Identical to BS_3STATE as far as the <i>control's</i> look is concerned and the issuing of the WM_CONTROL message to its owner window; but automatically changes its state when it is selected.
BS_USERBUTTON	7L	Creates a user-defined button that notifies its parent about its selection state; the request for <i>painting</i> through WM_PAINT is instead addressed to the owner window.
BS_BITMAP	0x0040L	The button displays a bitmap.
BS_ICON	0x0080L	The button displays an icon in place of the traditional text string.

*(continued)*

Table 7.5 (Continued)

<i>Style</i>	<i>Value</i>	<i>Description</i>
BS_HELP	0x0100L	Creates a button that causes the message WM_HELP to be issued when it is selected.
BS_SYSCOMMAND	0x0200L	Creates a button that issues the message WM_SYSCOMMAND when it is selected.
BS_DEFAULT	0x0400L	Creates a button with a heavy border, which indicates that the user is allowed to select the button by pressing the Enter key.
BS_NOPOINTERFOCUS	0x0800L	Creates a button that does not receive the <i>focus</i> even when it is selected.
BS_NOBORDER	0x1000L	Creates a button with no border.
BS_NOCURSORSELECT	0x2000L	If set together with BS_AUTORADIOBUTTON it prevents the button from being selected automatically when the user moves the cursor over that button through the direction keys.
BS_AUTOSIZE	0x4000L	The button takes on a different size from those assigned to it by the programmer if they are not large enough to contain the whole object.

Apart from the styles BS\_SYSCOMMAND and BS\_HELP that modify only the nature of the message generated by using windows of the WC\_BUTTON class, notice how different kinds of buttons can be reduced to *pushbuttons*, *checkboxes*, and *radiobuttons*, indicated by the styles BS\_PUSHBUTTON, BS\_CHECKBOX, and BS\_RADIOBUTTON. The style characterized by the word AUTO, as in BS\_AUTOCHECKBOX, indicates that the management of symbols used to represent selection, nonselection, and undetermined state (gray background in a *checkbox*), is automatically performed by PM over all of the windows that belong to a group. At the beginning of the development of Microsoft's user interfaces (in fact, PM is still influenced by the design of Windows, which guided its initial development phase in 1987), buttons were thought of as typical tools inside a dialog window. These windows, as we will see in the next chapter, are truly containers for controls of different classes. These controls are often grouped into homogenous areas by assigning to them the style WS\_GROUP. The idea of a group refers to this situation and use of buttons. With time, the evolution of user interfaces, usage rules, and conventions have changed. In this chapter we will limit ourselves to the windows of the class WC\_BUTTON as child windows of a client window.

The look of a traditional button, a pushbutton requires a three-dimensional look that is evident when the user "presses" the button with the mouse or the keyboard. The selection corresponds, in PM terminology, to being highlighted. On the other hand, checkboxes are rectangles with a text label to their right; radiobuttons are circles

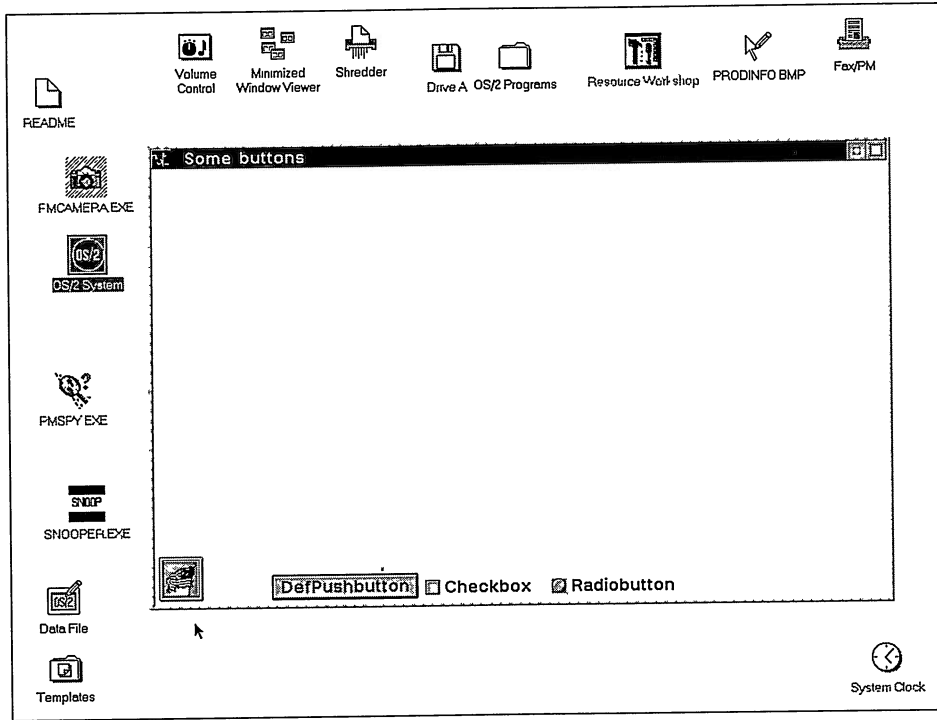
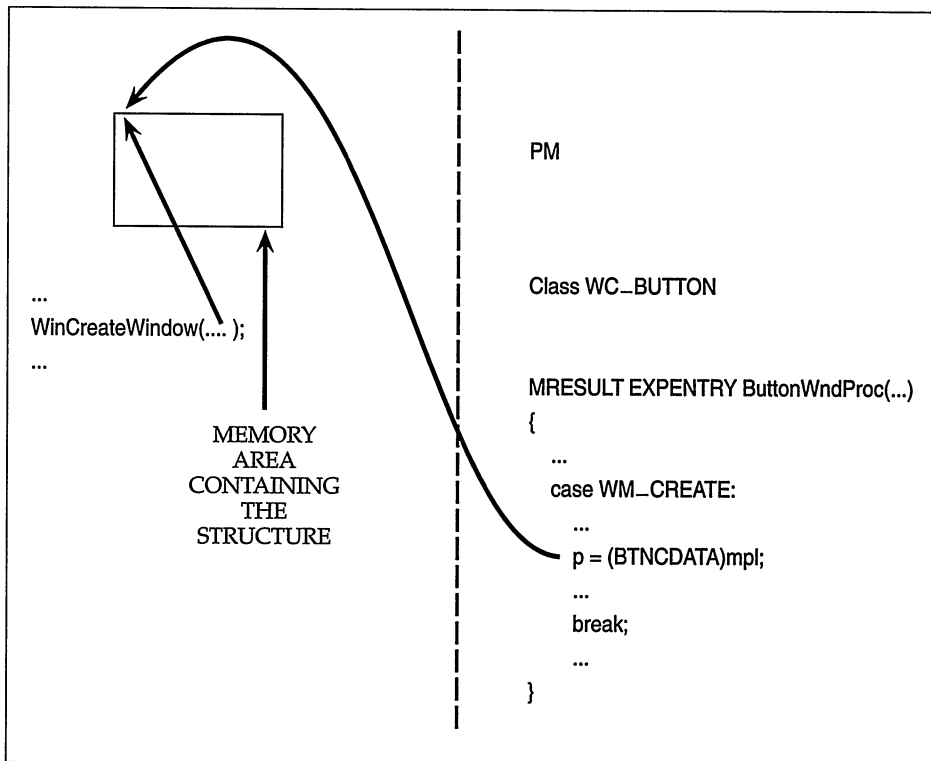


Figure 7.4 The look of buttons in Presentation Manager can be very different.

with a text label to their right. Listing 5.6, regarding the display of bitmaps and predefined pointers, gave you an idea of some of the possible looks that the `WC_BUTTON` class windows can have. Figure 7.4 is the result of Listing 7.2 in the creation of various kinds of buttons.

## Fashionable Buttons

By looking at Figure 7.4, you can see that a button is missing in the classical text, and replaced by an icon. This result is achieved by setting the style `BS_ICON` or `BS_BITMAP` if the image has larger dimensions. Drawn buttons are almost a *must* for applications nowadays. Just think of the *Smarticons* in Lotus's product line. Replacing a piece of text with an icon can enhance the interface of an application and make it easier for the user to interact with it. The syntax of `WinCreateWindow()` does not allow you to specify the handle of an icon or of a bitmap in any of its parameters. The obstacle can be overcome by passing to the next to last parameter the address of a `BTNCDATA` structure. This is a first example of a class specific data structure. When the window procedure of the class `WC_BUTTON` receives the message `WM_CREATE` in `mp1`, it will find exactly the same address you indicated when calling `WinCreateWindow()`. Figure 7.5 illustrates the whole process.



**Figure 7.5** Description of the mechanism used for passing information to a predefined window class.

### *The BTNCDATA Structure*

The structure `BTNCDATA` is specific to `WC_BUTTON` and allows the passing of class-specific data at the moment of creation of a window, independent of the `BS_` style that might possibly be set.

```
#define INCL_WINBUTTONS

typedef struct _BTNCDATA
{
    // btncd
    USHORT cb ;
    USHORT fsCheckState ;
    USHORT fsHiliteState ;
    ULONG hImage ;
} BTNCDATA ;
```

Have a look at the syntax of `BTNCDATA`. It is a nightmare! Compared to the past, a fourth member has been introduced, which is rightly of type `ULONG` like the system word. Inexplicably, for the first three members there has been no promotion from

USHORT to ULONG. So we are dealing with a hybrid object, with some related nontrivial problems.

The first member contains the size of the structure. The value returned by `sizeof` is that of a `long`; therefore, you need to apply a type casting correctly. The second and third members indicate together the selection state of the button. If it was specified during creation, you can avoid the subsequent issuing of the messages `BM_SETCHECK` and `BM_SETHILITE` to set the desired state.

The last member contains the handle of the icon or bitmap that you want to display in the button. Naturally, an assumption for the class `WC_BUTTON` to take into account is the presence of the flag `BC_ICON` or the flag `BS_BITMAP`. Here is how the result was achieved in Listing 7.2:

```
...
case WM_CREATE:
{
    HWND hwndButton ;
    HPOINTER hptr ;
    BTNCDATA btncdata ;

    btncdata.cb = (USHORT)sizeof( BTNCDATA) ;
    btncdata.fsCheckState = TRUE ;
    btncdata.fsHiliteState = TRUE ;
    btncdata.hImage = WinLoadPointer(  HWND_DESKTOP,
                                     NULLHANDLE, RS_ICON) ;

    hwndButton = WinCreateWindow(  hwnd, WC_BUTTON,
                                  NULL,
                                  WS_VISIBLE | BS_PUSHBUTTON |
                                  BS_AUTOSIZE | BS_ICON,
                                  5, 5, 0, 0,
                                  hwnd, HWND_TOP,
                                  CT_PUSHBUTTON,
                                  &btncdata, NULL) ;

    ...
}
```

The style `BS_AUTOSIZE` is quite handy. The window will be sized automatically, and you don't have to be concerned about it. Be careful, though! The style will work only if the extensions on the two axes are equal to 0. If they are not, you will obtain a window the size of which is equal to what you have indicated in `cx` and `cy`, despite the presence of `BS_AUTOSIZE`.

A limit to this solution is to be found in the lack of a three-dimensional effect on the image. The 3D look is the consequence of the presence of two different icons that differ one from the other only in the shadow of the lower and the right borders. When the button is pressed, the second icon is displayed in place of the first one, and the first one is restored when the button is released. There is no specific `BM_` message to dynamically change the icon shown in a button. The only way to achieve this is by means of `WM_SETWINDOWPARAMS`. This message is quite powerful and flexible, as seen in Listing 7.2.

Another solution to create true 3D buttons requires you to declare them with the style `BS_USERBUTTON`. The buttons equipped with `BS_USERBUTTON` will delegate to their owner performing the appropriate painting operations: that is, in this case, handling a pair of related icons, which is quite easy, as you will see when we will explore listboxes.

## Interacting with a Button

Buttons of the class `WC_BUTTON` dispose of some specific messages for their own needs, as summarized in Table 7.6. Among these, the most frequently used is `BM_SETCHECK`, which will let you set/remove the selection state on a checkbox or a radiobutton. Generally, the interaction that takes place with a traditional pushbutton will involve the application's intercepting the notification codes generated by the class's window procedure and addressed to the owner window.

**Table 7.6 Messages That Are Specific to the Class `WC_BUTTON`**

<i>Message</i>	<i>Value</i>	<i>Description</i>
<code>BM_CLICK</code>	0x0120	Issued to simulate the pressing of the button.
<code>BM_QUERYCHECKINDEX</code>	0x0121	This message will allow you to identify the selected <i>radiobutton</i> within a group, and will return its position with an index that starts counting from zero.
<code>BM_QUERYHILITE</code>	0x0122	Asks a button to return its selection state: returns TRUE if it is selected.
<code>BM_SETHILITE</code>	0x0123	Assigns/removes the selection state to/from a button.
<code>BM_QUERYCHECK</code>	0x0124	Asks a button if it is displaying its selection symbol: can be sent to buttons defined with the styles <code>BS_CHECKBOX</code> , <code>BS_AUTOCHECKBOX</code> , <code>BS_3STATE</code> , <code>BS_AUTO3STATE</code> , <code>BS_RADIOBUTTON</code> , and <code>BS_AUTORADIOBUTTON</code> : returns 1 if the button is selected, 2 if it is not selected, and 0 if its state is undetermined.
<code>BM_SETCHECK</code>	0x0125	This message will let you set the state of a button: pressed or not pressed.
<code>BM_SETDEFAULT</code>	0x0126	Assigns/removes to/from a <i>pushbutton</i> or a <i>userbutton</i> the state of default button.



**Table 7.7 The Notification Codes of the Class WC\_BUTTON**

<i>Notification Code</i>	<i>Value</i>	<i>Description</i>
BN_CLICKED	1	Indicates that you have pressed the mouse button over a <i>radiobutton</i> or over a <i>checkbox</i> .
BN_DBLCLICKED	2	Indicates that you have double-clicked the mouse button over a <i>radiobutton</i> or over a <i>checkbox</i> .
BN_PAINT	3	Instructs the application to perform the <i>painting</i> of a button created with the style BS_USERBUTTON.

## *Notification Codes*

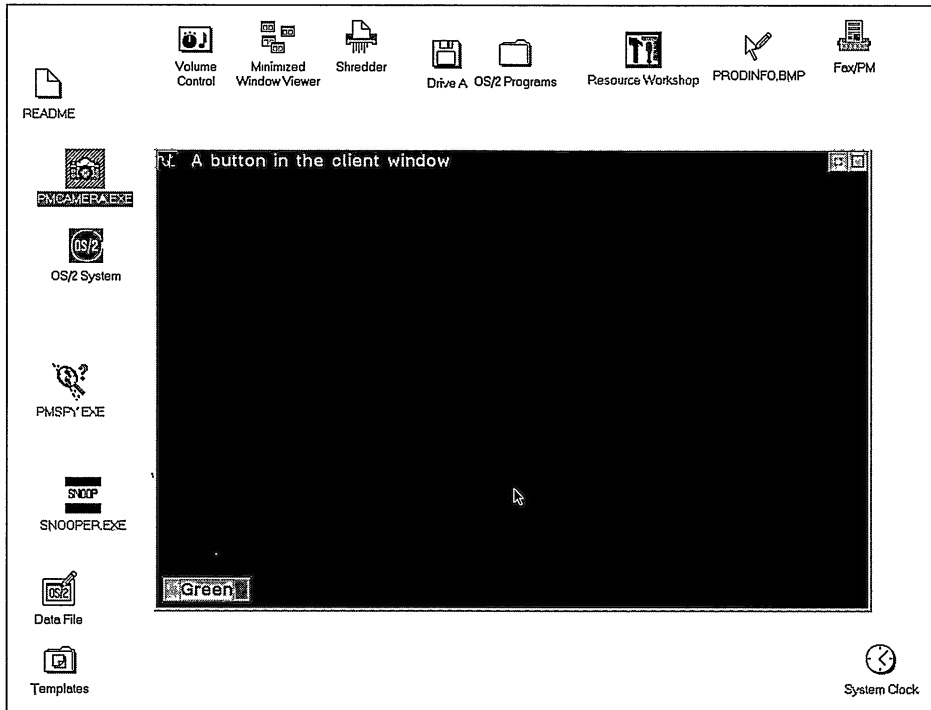
The buttons of the class WC\_BUTTON are unusual in that some of them communicate with their parent/owner window by sending the message WM\_CONTROL, while others use the message WM\_COMMAND. The distinction between the two kinds is in the type of button that was created, that is, in the particular BS\_ style adopted. Pushbuttons will issue the WM\_COMMAND message, while checkboxes and radiobuttons will use WM\_CONTROL. The pushbuttons' use of WM\_COMMAND explains the condition CMDSRC\_PUSHBUTTON that we saw in Chapter 6 when analyzing the source of the WM\_COMMAND messages. Pushbuttons use this means for communicating with their parent/owner window. The notification codes issued by the window procedure of the class WC\_BUTTON are summarized in Table 7.7.

Since both radiobuttons and pushbuttons are tools used in groups, their use is concentrated almost exclusively in dialog windows. The interception of the message WM\_CONTROL containing the notification codes takes place at the *dialog procedure* level. We will examine in great detail these two kinds of buttons in the next chapters, with significant code samples.

## *Pushbuttons as Input Elements*

A pushbutton's issuing the WM\_COMMAND message to its parent/owner window, just as with menus and accelerators, suggests a simple consideration. Both menus and accelerators are typical input tools that allow the user to interact easily with the application. In some cases it is possible to use buttons in addition to or in place of menus, thus giving the window a very appealing look—while remaining faithful to the rules of CUA—by implementing a sound and ergonomic input tool. That is precisely what we intend to do with the next listing (Listing 7.3).

We have already faced the problem of changing the client window's color in many places in this text. Now we want to propose a solution that is simple, practical, and convenient, both for the software designer and the user. This solution is based on pushbuttons. It is not unusual but it will give you some idea as to how the pushbuttons are not limited to use in dialog windows.



**Figure 7.6** Placement of a button directly in the client area of a window where it is used as an input tool.

In the client window of an application there will be a pushbutton with the label Green that allows the user to indicate the color to be used for painting the window (Figure 7.6).

The selection of the pushbutton with the mouse will switch the window to the next available color (Blue). In the application's code, the message `WM_COMMAND` is intercepted because the application has a menu bar. Before extracting, with the macro `COMMANDMSG`, the ID of the window that generated the message, the same macro `COMMANDMSG` is used to identify the source of the message:

```
...
case WM_COMMAND :
    switch( COMMANDMSG( &msg) -> source)
    {
        case CMDSRC_MENU:
            switch( COMMANDMSG( &msg) -> cmd)
            {
                default:
                    break ;
            }
        break ;
    }
break ;
```

```

case CMDSRC_PUSHBUTTON:
    switch( COMMANDMSG( &msg) -> cmd)
    {
        case CT_DEFPUSHBUTTON:
            ( nCnt == 2) ? nCnt = 0 : ++nCnt ;

            WinSetWindowText( hwndButton, szClr[ nCnt] ) ;
            WinInvalidateRect( hwnd, NULL, FALSE ) ;
            break ;

        default:
            break ;
    }
    break ;
}
break ;
...

```

The code fragment dealing with CT\_DEFPUSHBUTTON is very simple—first the counter `nCnt` is incremented or zeroed by using the conditional operator. Then the text to be displayed in the pushbutton is modified with `WinSetWindowText()` and the text string that is retrieved from the array of pointers `szClr[]` corresponds to the next color. To change the color in the client window immediately, the whole area is invalidated and the sending of a WM\_PAINT is forced. Another valid solution would have been to use the style CS\_SYNCPAINT at the window class registration level. In this way—in Listing 7.3 this approach to the problem is presented—it is possible to avoid calling `WinUpdateWindow()`.



A true PM application might accommodate several buttons in the client window, with much more interesting designs. A sound approach consists of aligning the various buttons at the bottom of the window in order to be able to perform all operations directly and easily. For instance, in a communication program or an e-mail service program, you might include buttons like Send, Receive, and so on. A collection of adjacent buttons is functionally similar to the *SmartIcons* bar typical of Lotus's line of products.

A nice touch would then be the addition of specific accelerators for these pushbuttons. In fact, a mouseless user wouldn't have a chance to select this kind of button, so accelerators are an ideal solution. We have already seen in Listing 6.9 of the previous chapter how to dynamically implement an accelerator table through combined calls to `WinLoadAccelTable()` and `WinSetAccelTable()`. This task is left to you as an exercise (you will find the solution in Chapter 8).

---

## The Class WC\_STATIC

In Chapter 5, when dealing with keyboard handling, we used the function `GpiCharStringAt()` to display a text string inside a window. This operation required you to have at hand a handle and a presentation space, a typical tool for *Gpi* functions, and

for some *Win* functions. An alternative solution, valid under certain circumstances, is to create a window of class `WC_STATIC`. Then, the output would be managed through `WinSetWindowText()`, while text retrieval would be accomplished through `WinQueryWindowText()`.

Actually, in all of the samples seen so far, the need to display text in the client window has been rather limited. Resorting to a window of class `WC_STATIC` is more usual for dialog windows (Chapter 8), windows that feature a large number of controls, and that are indicated for interacting directly with the user. It is in this context that windows of this class get used most often. Let's examine the dialog window to open a file in the system editor `E.EXE` (Figure 7.7). The labels `Open`, `File Name`, `Files`, `Directories`, and `Directory` are actually windows of the class `WC_STATIC`.

## The `SS_Styles`

There should exist a limited number of different styles to qualify the window of the class `WC_STATIC`, due to the reduced number of user interactions supported (this class does not provide the user with any input capability). Presentation Manager takes advantage of this class to perform other actions in addition to simply displaying text statically. Therefore, there are as many as 12 different styles (Table 7.8) that characterize the class `WC_STATIC`.

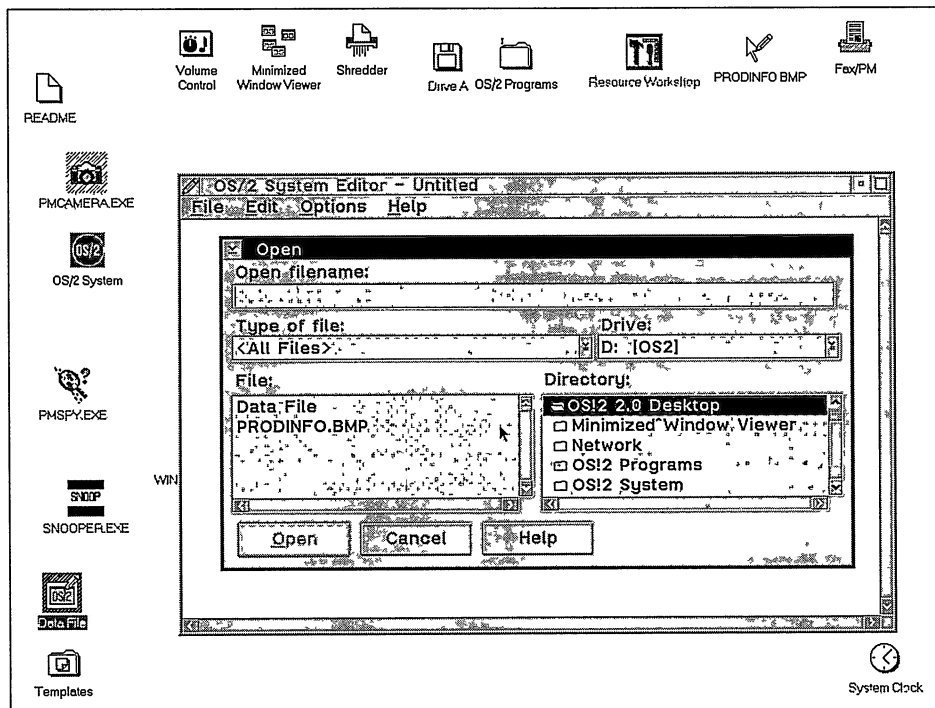


Figure 7.7 The dialog window to open a file in the system editor.

By examining Table 7.8, it is evident that the `SS_` styles are mutually exclusive. This means that it is not possible to create a static text window that, at the same time, has a light gray frame around it. A window of class `WC_STATIC` can play only one role at a time.

A text window surrounded by a frame, however, is quite often needed. To solve this problem I noticed that a PM application used to install some printer drivers drew a rectangle around the text. There is a simpler solution, based on the services offered by the class `WC_ENTRYFIELD`.

You might be wondering why you should resort to the services of the class `WC_STATIC` to display an icon or a bitmap, when the operation was performed

**Table 7.8 The Styles of the Class `WC_STATIC`**

<i>Style</i>	<i>Value</i>	<i>Description</i>
<code>SS_TEXT</code>	0x0001L	This style is typically used to create windows for displaying static text.
<code>SS_GROUPBOX</code>	0x0002L	Creates a window surrounded by a light frame and a text string in the upper left-hand corner: this kind of window is used inside a <i>dialog window</i> to group together related <i>controls</i> , like <i>radiobutton</i> .
<code>SS_ICON</code>	0x0003L	Creates an icon by interpreting the text (given as the window's title) as the ID of the resource that is to be loaded from the resource file.
<code>SS_BITMAP</code>	0x0004L	Creates a bitmap according to the same rules followed for <code>SS_ICON</code> .
<code>SS_FGNDRECT</code>	0x0005L	Creates a rectangle filled with the foreground color.
<code>SS_HALFTONERECT</code>	0x0006L	Creates a rectangle filled with light gray.
<code>SS_BKGNDRECT</code>	0x0007L	Creates a rectangle filled with the background color.
<code>SS_FGNDFRAME</code>	0x0008L	Creates a rectangle with a frame colored in the foreground color.
<code>SS_HALFTONEFRAME</code>	0x0009L	Creates a rectangle with a frame colored in light gray.
<code>SS_BKGNDFRAME</code>	0x000aL	Creates a rectangle with a frame color in the background color.
<code>SS_SYSICON</code>	0x000bL	Creates a system icon by interpreting the text given to the parameter <code>pszName</code> of <code>WinCreateWindow()</code> as the ID of the system icon.
<code>SS_AUTOSIZE</code>	0x0040L	Automatically sizes the window so it can contain the related object.

**Table 7.9 The Messages of the Class WC\_STATIC**

<i>Message</i>	<i>Value</i>	<i>Description</i>
SM_SETHANDLE	0x0100	Issued to set the handle of an icon or a bitmap in a window of class WC_STATIC.
SM_QUERYHANDLE	0x0101	Issued to retrieve the handle of an icon or a bitmap in a window of class WC_STATIC.

successfully in more than one of the previous examples. Good question! The answer is simple, and always the same: The class WC\_STATIC becomes useful for displaying an icon or a bitmap in a *dialog window*. Just a little more patience, and you will discover what this window is all about!

### The SM\_ Messages

The set of messages specific to the class WC\_STATIC is very small, and pertains only to this class as a container of graphical objects, like bitmaps and icons (Table 7.9).

It is easy to infer that since there is no possibility for the user to interact with a window of class WC\_STATIC, there are no special notification codes.

---

## The Class WC\_TITLEBAR

The titlebar of a generic PM application belongs to the class WC\_TITLEBAR. Its main purpose is that of giving the place for the application window's title. Other duties that are also important are moving the window when the user is pressing the left mouse button while sliding the mouse pointer on the screen, and changing the window's size when detecting a double-click.

Moving a window in the space of its parent is something solved through the titlebar by calling the function *WinTrackRect()*, which we have already encountered in Chapter 5, when exploring the mechanisms used for moving a bitmap in a window's client area.

Resizing (maximizing or restoring) a frame window is an operation activated through a double-click on the titlebar. If the window is maximized, it will be restored to its previous size and position within the space of its parent window; otherwise it will be maximized. This will always be dealt with by intercepting the message WM\_BUTTONXDBLCLK in the class's window procedure.

The assignment of a text label to the titlebar is performed directly when creating the window or later through *WinSetWindowText()*, though without specifying directly the handle of this window; the frame's handle is more than adequate. To retrieve the title's text you can call the function *WinQueryWindowText()*.

**Table 7.10 Messages of the Class WC\_TITLEBAR**

<i>Message</i>	<i>Value</i>	<i>Description</i>
TBM_SETHILITE	0x01e3	Sets the state of the titlebar.
TBM_QUERYHILITE	0x01e4	Retrieves the state of the titlebar.

There are no special flags for this class, due to the simple and limited nature of the operations. The messages, instead, are only two, as listed in Table 7.10.

The titlebar can show up in two different conditions: activated (*highlighted*) or deactivated. By default, these two different conditions are represented respectively with the colors green and gray. In general the color attribute of the titlebar varies according to the presence or absence of the input focus over a window. By sending TBM\_SETHILITE with mp1 set to FALSE to the active window, you change the titlebar's color from green to gray without changing the relationship between the window and the keyboard.

---

## The Class WC\_SCROLLBAR

One of the principal advantages when adopting a graphical user interface is that of easily and effectively implementing output *scrolling*, both vertically and horizontally. The class WC\_SCROLLBAR provides the software designer with an excellent tool for making it easier for the user to perform these operations. The windows of the class WC\_SCROLLBAR do not actually perform the scrolling. They are simply a graphical tool used on the screen to control a scrolling operation. It is always the application's duty to be clever enough to "understand" how much scrolling is being requested by the user and to perform proportionally the true scrolling in the window with which the scrollbar is associated.

### *The SBS\_ Styles*

The look of a scrollbar is standardized and cannot be changed by the software designer. Instead, what needs to be decided is the direction of scrolling. Table 7.11 summarizes all styles of the class WC\_SCROLLBAR.

### *The SBM\_ Messages*

Positioning a scrollbar inside a window is a simple operation. In addition to the classic approach of creating a window of this class by calling *WinCreateWindow()*, you can also use the frame control flags FCF\_HORZSCROLL and FCF\_VERTSCROLL that perform all operations automatically. A frame window is, in fact, ready for receiving two controls of the class WC\_SCROLLBAR, thanks to the existence of the predefined IDs FID\_VERTSCROLL and FID\_HORZSCROLL.

**Table 7.11 The Styles of the Class WC\_SCROLLBAR Allow You to Define a Tool for Controlling Horizontal or Vertical Scrolling**

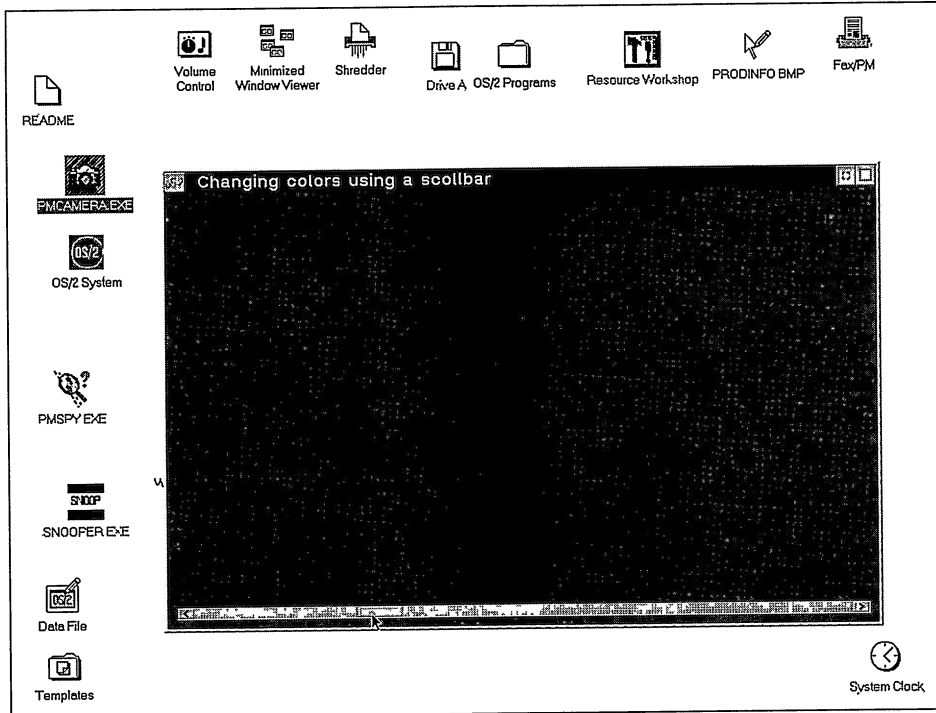
<i>Style</i>	<i>Value</i>	<i>Description</i>
SBS_HORZ	0x0000L	Creates a horizontal <i>scrollbar</i> .
SBS_VERT	0x0001L	Creates a vertical <i>scrollbar</i> .
SBS_THUMBSIZE	0x0002L	Indicates the presence of numeric values in the members <i>cVisible</i> and <i>Ctotal</i> of the class's <i>SBCDATA</i> structure.
SBS_AUTOTRACK	0x0004L	The position indicator moves in the <i>scrollbar</i> when scrolling actually takes place.
SBS_AUTOSIZE	0x2000L	The scroll indicator changes size according to the amount of actual scrolling.

Whatever method is used for creating a scrollbar, you need to define the range of scrolling by setting a lower and an upper limit for the position identifier (*slider*). Both of these quantities have to be specified by the application before you can use the scrollbar; this ensures that the programmer is always in control. It is the message *SBM\_SETSCROLLBAR* that performs this task (Table 7.12).

**Table 7.12 The Messages of the Class WC\_SCROLLBAR**

<i>Message</i>	<i>Value</i>	<i>Description</i>
<i>SBM_SETSCROLLBAR</i>	0x01a0	Defines the <i>slider's</i> position and the range of scrolling.
<i>SBM_SETPOS</i>	0x01a1	Defines the <i>slider's</i> position in the <i>scrollbar</i> .
<i>SBM_QUERYPOS</i>	0x01a2	Returns the <i>slider's</i> position in the <i>scrollbar</i> .
<i>SBM_QUERYRANGE</i>	0x01a3	Returns the scrolling range.
<i>SBM_SETTHUMBSIZE</i>	0x01a6	Defines the size of the <i>slider</i> by indicating the total number of elements represented by the <i>scrollbar</i> and how many of these are visible. The <i>slider's</i> size is computed on the basis of these values; changing the size of the <i>slider</i> makes it easier to infer the size of the object that is being examined.
<i>WM_HSCROLL</i>	0x0032	Notifies the <i>scrollbar's</i> owner about the type and amount of horizontal movement that needs to be applied to the <i>slider</i> .
<i>WM_VSCROLL</i>	0x0031	Notifies the <i>scrollbar's</i> owner about the type and amount of vertical movement that needs to be applied to the <i>slider</i> .





**Figure 7.8** A horizontal scrollbar controls the color selection for the client window through the movements of the slider.

Let's now get back to the example of coloring the client area of a window; this time, though, we will allow the user to select a color through a scrollbar. Figure 7.8 presents the program's output.



The source code of SCROLLBAR is presented in Listing 7.4. The scrollbar is created in the window procedure of the client window in the code fragment dealing with the message WM\_CREATE. Its position, with respect to the parent window, is performed partially in the code of WM\_CREATE and partially in WM\_SIZE. The size on the X axis can vary according to possible changes in the main window's size.

Immediately after creating the scrollbar, SCROLLBAR sets the range and the position of the slider. Both are done sending a message directly to the WC\_SCROLLBAR class window. The limits of 1 and 15 correspond to the CLR\_color IDs, as defined in PMGPL.H.

```

...
WinSendMessage(   hwndhScroll, SBM_SETSCROLLBAR,
                  MPFROMSHORT( 0),
                  MPFROM2SHORT( 1, 15) );
...

```

An alternative solution would be that of passing this initialization data and other data when the window is created, by taking advantage of the structure `SBCDATA`, as we will see shortly.

Much more interesting is examining the code regarding the notification of the actions performed by the user with the scrollbar. There are no proper notification codes for this class; instead, the class issues the messages `WM_HSCROLL` and `WM_VSCROLL` in order to notify the owner of the behavior, respectively, of a horizontal scrollbar and a vertical one.

The message `WM_HSCROLL` contains in `mp1` the ID of the control that is performing the notification, while `mp2` packs the code that indicates the action performed and the position of the slider. The following code fragment shows some possible values for a horizontal scrollbar.

```
...
case WM_HSCROLL:
    switch( SHORT2FROMMP( mp2))
    {
        case SB_LINELEFT:
            sPos -= 1 ;
            break ;

        case SB_LINERIGHT:
            sPos += 1 ;
            break ;

        case SB_PAGELEFT:
            sPos -= 3 ;
            break ;

        case SB_PAGERIGHT:
            sPos += 3 ;
            break ;

        case SB_SLIDERPOSITION:
            sPos = SHORT1FROMMP( mp2) ;
            break ;
    }
...

```

Each `SB_` case examines a possible user action. The first two regard the pressing of the left mouse button over the left or right arrow that is positioned at the scrollbar's extremities. In this case, the application has determined that the movement of the slider should be one unit in both directions. The third and fourth cases correspond to the pressing of the left mouse button directly over the scrollbar in the area between one extremity and the slider. This second condition is implemented in the application by moving three units, which correspond to selecting the three previous or following colors. To update the position of the slider, send the message `SBM_SETPOS`:

```
...
WinSendMsg( hwndScroll, SBM_SETPOS,
            MPFROMSHORT( sPos), 0L) ;
...

```

The logic just described for a horizontal scrollbar can be extended to a vertical one. In this case, the values contained in `SHORT2FROMMP` (mp2) can take on the syntax of `SB_LINEUP`, `SB_LINEDOWN`, `SB_PAGEUP`, and `SB_PAGEDOWN`.

In the example, the part played by the scrollbar is somewhat analogous to that of an input tool. If, however, the goal is to scroll a piece of text or a bitmap, it is the application's responsibility to call the function `WinScrollWindow()` to perform the movement in the given direction.

The case `SB_SLIDERPOSITION` will restore the slider's position after the user has dragged it by keeping the left mouse button down over the slider. If you need to track the slider's position at every instant, then you must intercept the code `SB_SLIDERTRACK` and update and display the information dynamically.

## Some Considerations

An appealing aspect of OS/2 PM's scrollbars is automatic handling of the scroll range in both directions. When you reach one end of the scrollbar, the scrollbar itself will disable any further scrolling in that direction, and thus the application does not need to check the limits that might possibly have been reached by the user interaction.

Often, windows of the class `WC_SCROLLBAR` are inserted automatically in other windows, like *listboxes* or *mles* simply by setting a style flag. Their behavior is always similar to that just described, with the difference that their owner is—this time—another control.

The class `WC_SCROLLBAR` has a specific data structure in `PMWIN.H` which is used to pass specific information when a window is created.

```
#define INCL_WINSCROLLBARS
typedef struct _SBCDATA
{ // sbcd
    USHORT cb ;
    USHORT sHilite ;
    SHORT posFirst ;
    SHORT posLast ;
    SHORT posThumb ;
    SHORT cVisible ;
    SHORT cTotal ;
} SBCDATA ;

typedef SBCDATA *PSBCDATA ;
```

By filling in the members of this structure you will not have to later define the slider's position (`posThumb`) and the scroll range (`posFirst` and `posLast`). The address of the `SBCDATA` structure is passed to the function `WinCreateWindow()` as its twelfth parameter, as you can see in the following example:

```
...
SCBDATA sbcd ;

sbcd.cb = sizeof sbcd ;
sbcd.sHilite = 0 ;
```

```

sbcd.posFirst = sbcd.posThumb = 1 ;
sbcd.posLast = sbcd.cTotal = 100 ;
sbcd.cVisible = 10 ;
sbcd.posFirst = 0 ;
sbcd.posLast = 100 ;

hwndScroll = WinCreateWindow( hwnd, WC_SCROLLBAR, NULL,
                               WS_VISIBLE | SBS_HORZ,
                               10, 10, 18, 120,
                               hwnd, HWND_TOP, CT_HSCROLL,
                               &sbcd, NULL) ;

...

```

## The Class WC\_LISTBOX

This window class is probably one of the most appreciated by software designers, thanks to its flexibility. Listboxes are found in a number of dialog windows, and even in notebook pages for setting some operating features of WPS objects. They display text strings in a limited area of the screen, and they scroll their output vertically. This description is not complete. For example, scrolling is supported in both directions. Also, the objects contained in a listbox don't have to be text strings only, but can also be icons or bitmaps or any combination of these (Figure 7.9).

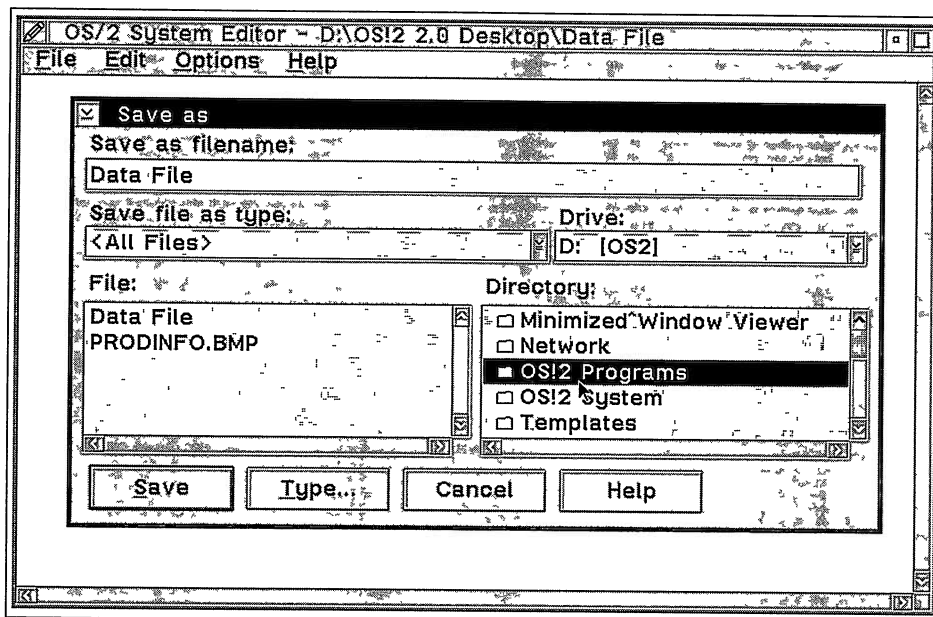


Figure 7.9 A typical dialog window for saving a file contains more than a listbox.

**Table 7.13 The Styles of the Class WC\_LISTBOX**

<i>Style</i>	<i>Value</i>	<i>Description</i>
LS_MULTIPLESEL	0x00000001L	Defines a multiple selection <i>listbox</i> .
LS_OWNERDRAW	0x00000002L	Delegates to the application the task of <i>painting</i> each item present in the <i>listbox</i> .
LS_NOADJUSTPOS	0x00000004L	Allows the vertical dimension of the <i>listbox</i> not to be an exact multiple of the height of the current system font.
LS_HORZSCROLL	0x00000008L	Adds a horizontal <i>scrollbar</i> in addition to the vertical one that is always present.
LS_EXTENDEDSEL	0x00000010L	Creates an extended selection <i>listbox</i> .

## The Styles of the Class WC\_LISTBOX

Despite the fact that a window of class WC\_LISTBOX can do many different things, the styles available for it are limited in number (Table 7.13).

Thus, quite often, to create a listbox you only need to specify the name of the class, with no special style. However, it is probably better to use LS\_NOADJUSTPOS so that the listbox can be extended vertically without depending on the height of the system font being used. In fact, a listbox will generally change its own vertical size so that it corresponds to an exact multiple of the system font.

It is good design to implement a horizontal scrollbar only if strictly necessary. In general, scrolling is just a chore for the user, barely acceptable on the vertical axis, almost useless when it takes place horizontally, because the information is difficult to read. The selection of several items is enabled by setting the flag LS\_MULTIPLESEL or LS\_EXTENDEDSEL. The advantage of the second solution is that you can perform a *swipe* selection directly with the mouse. Instead, with LS\_MULTIPLESEL, a selection pertains to only one item at a time. The styles are completed by LS\_OWNERDRAW, which lets you create listboxes the output of which is handled by the application, and not by the class WC\_LISTBOX.

## The LM\_Messages

To insert, add, select, or remove a text item in a listbox, you have available 15 different messages, all characterized by the prefix LM\_ (Table 7.14).

In practice, all the messages listed in Table 7.14 pertain to the management of the text in a listbox. The insertion of graphical objects, like icons or bitmaps, requires a greater control over the behavior of a listbox, and this can be achieved only by delegating to the application the management of output in the window. In that case it is necessary to set the style LS\_OWNERDRAW. The complex nature of a listbox requires a thorough understanding of the terminology used for describing the messages in Table 7.14. To this end, let's examine Figure 7.10.

**Table 7.14 The Messages of the Class WC\_LISTBOX**

<i>Message</i>	<i>Value</i>	<i>Description</i>
LM_QUERYITEMCOUNT	0x0160	Returns the number of items present in a <i>listbox</i> .
LM_INSERTITEM	0x0161	Inserts a new item into a <i>listbox</i> . The operation can take place in different ways: appending the new item to the end of the list, or ordering items in ascending or descending ASCII order.
LM_SETTOPINDEX	0x0162	Performs vertical <i>scrolling</i> of the contents of the <i>listbox</i> so that the indicated item will appear in the first visible position.
LM_DELETEITEM	0x0163	Deletes a specific item from the <i>listbox</i> by specifying its index.
LM_SELECTITEM	0x0164	Selects an item from the <i>listbox</i> by specifying its index.
LM_QUERYSELECTION	0x0165	Returns the index of the selected item, starting from the origin or from the item previously selected if this is a multiple selection <i>listbox</i> .
LM_SETITEMTEXT	0x0166	Replaces the text of an item with what is present in the buffer.
LM_QUERYITEMTEXTLENGTH	0x0167	Returns the length of the text of an item.
LM_QUERYITEMTEXT	0x0168	Returns and stores in a buffer the text of an item.
LM_SETITEMHANDLE	0x0169	Stores the value of an item in the reserved memory area.
LM_QUERYITEMHANDLE	0x016a	Retrieves the value of an item stored in the reserved memory area.
LM_SEARCHSTRING	0x016b	Searches for a text string in all items of the <i>listbox</i> .
LM_SETITEMHEIGHT	0x016c	Sets the height of each item in the <i>listbox</i> .
LM_QUERYTOPINDEX	0x016d	Returns the index, relative to the origin, of the topmost item in the <i>listbox</i> .
LM_DELETEALL	0x016e	Empties the <i>listbox</i> of its contents.

As you have seen in the class WC\_SCROLLBAR, the vertical scrollbar of a listbox automatically handles the activation/deactivation of the scroll icons according to the number and position of items present in the listbox. The first visible item in the listbox is known as the *top-index*. The scrollbar has the listbox both as its parent and owner.

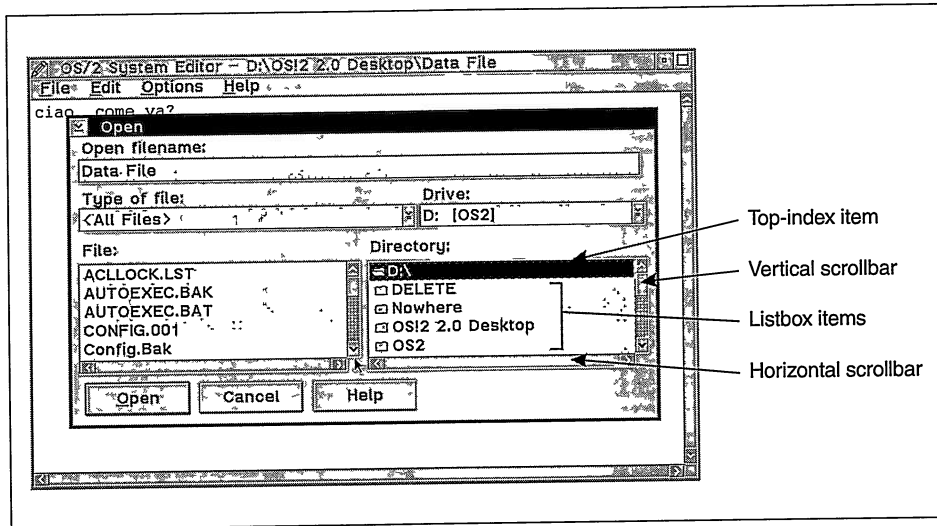


Figure 7.10 The structural elements of a listbox.

Each item is characterized by an ID that is automatically assigned by the listbox on the basis of the item's position within the list. The numbering scheme always starts from zero for the first existing menu item. Messages are sent to the listbox, and almost all of them require you to indicate the ID of the item affected by the action to be performed. To simplify some operations, the system provides specific macros, listed in Table 7.15. Each of these macros returns a data type appropriate for the corresponding and implied LM\_ message.

Table 7.15 Macros Used to Interact with Listboxes

Macro	Description
<code>LONG WinDeleteLboxItem( hwndLbox, index );</code>	Deletes an item from the <i>listbox</i> .
<code>LONG WinInsertLboxItem( hwndLbox, index, psz );</code>	Inserts an item into the <i>listbox</i> and specifies how to insert it.
<code>LONG WinQueryLboxCount( hwndLbox );</code>	Returns the number of items present in the <i>listbox</i> .
<code>LONG WinQueryLboxItemText( hwndLbox, index, psz, cchMax );</code>	Returns the text of an item.
<code>SHORT WinQueryLboxItemTextLength( hwndLbox, index );</code>	Returns the length of the text of an item.
<code>BOOL WinSetLboxItemText( hwndLbox, index, psz );</code>	Sets the text of an item.
<code>LONG WinQueryLboxSelectedItem( hwndLbox );</code>	Returns the index of the selected item in the <i>listbox</i> .







```

{
    ...
    sStart = sPos ;
}
...

```

The identifier `sStart` is initially equated to `LIT_FIRST`, in order to identify the first physical item. The return value of the call to `WinSendMsg()` is stored in the identifier `sPos`, and corresponds to the selected item.

After performing the application-specific operations with that item, assign to `sStart` the value of `sPos` and repeat the search, this time starting from the point identified in the previous step. The `while` loop will end when the request for a new selected item returns the value of `LIT_NONE`.

In some situations a multiple selection listbox can behave oddly (as you will see when writing the utility `WHEREIS.EXE`, in Chapter 8). The solution is to display the first item as the window's top-index by sending the message `LM_SETTOPINDEX`:

```

...
WinSendMsg( hwndList, LM_SETTOPINDEX, MPFROMSHORT( 0), 0L );
...

```

The first index of a listbox is always characterized by the value of 0.

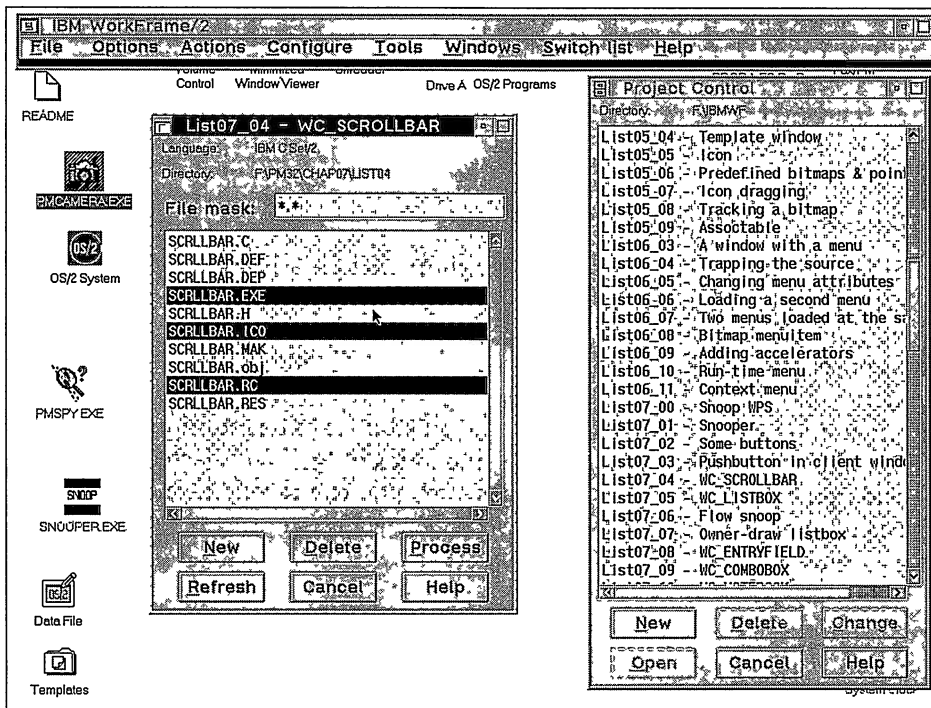


Figure 7.12 A multiple selection listbox.

**Table 7.16 The Notification Codes Sent from a Listbox to the Window Procedure of the Owner Window**

<i>Notification Code</i>	<i>Value</i>	<i>Description</i>
LN_SELECT	1	The user has selected an item from the list.
LN_SETFOCUS	2	The <i>input focus</i> has been transferred to the <i>listbox</i> .
LN_KILLFOCUS	3	The <i>input focus</i> has left the <i>listbox</i> .
LN_SCROLL	4	The user has scrolled the contents of the <i>listbox</i> .
LN_ENTER	5	The user pressed the Enter key or performed a double-click with the mouse over an item.

## The Notification Codes

The actions notified by a listbox to its owner concern various kinds of interactions between the user and the window: the selection of an item, selection and acceptance of an item (double-click), and vertical scrolling of the contents of the listbox. The notification codes used by the class WC\_LISTBOX are listed in Table 7.16 and are reported by issuing the message WM\_CONTROL.

WM_CONTROL	0x0030	<i>Description</i>
mp1	USHORT id USHORT usnotifycode	ID of the <i>control</i> . Notification code.
mp2	ULONG ulcontrolspect	Notification code specific and variable information.
Return Value	ULONG flreply	Reserved.

With WM\_CONTROL the predefined classes notify the owner about the action that took place over a window of that class. mp1 will always contain the same packed information, independent of the source that issued the message: the ID of the window that issued the WM\_CONTROL, and then the notification code.

The whole of mp2 will often contain a pointer to the data area used to convey yet more information to the owner. At other times, mp2 will simply contain the handle of the control.

A typical scheme for intercepting the WM\_CONTROL message in window procedure of the listbox's owner window (or in some other predefined window), is the following:

```

...
case WM_CONTROL:
    switch( SHORT1FROMMP( mp1))
    {
        case CT_LISTBOX:
            switch( SHORT2FROMMP( mp1))
            {

```

```

        case LN_ENTER:
            ...
            break ;

        case LN_SELECT:
            ...
            break ;

    }
    break ;
    ...

}
break ;
...

```

In each case branch identifying a notification code, the programmer usually sends one or more messages to the listbox in order to retrieve the information it needs. For instance, if the code were `LN_ENTER`, then you would be interested in knowing which item was selected and getting the corresponding text. So you must first obtain the index, and you can do this by issuing the message `LM_QUERYSELECTION`, and then retrieving the text with `LM_QUERYITEMTEXT`:

```

...
case LN_ENTER:
{
    SHORT sPos ;
    CHAR szString[ 20] ;

    sPos = (SHORT)WinSendMsg(   HWNDFROMMP( mp2), LM_QUERYSELECTION,
                                MPFROMSHORT( LIT_FIRST), 0L) ;

    WinSendMsg(   HWNDFROMMP( mp2), LM_QUERYITEMTEXT,
                MPFROM2SHORT( sPos, sizeof szString),
                MPFROMP( szString)) ;

    ...
    break ;
...

```

In the case of notification codes issued by a window of class `WC_LISTBOX`, you will find the window's handle in `mp2`.

## *A Simple Listbox*

We will make heavy use of listboxes in all of the following chapters. It's time to get to know how this window works in practice: We start by creating a simple application that displays in a listbox positioned inside the client area of a top-level window all Marriott hotels in which the author has stayed in recent years. The text strings reside

in the STRINGTABLE area of the resource file, and their IDs are ordered numerical values. In addition to the listbox, the application also creates a window of class WC\_STATIC, and places it above the listbox. Both operations take place when the message WM\_CREATE is intercepted in the window procedure of the main window:

```

...
case WM_CREATE:
{
    hwndStatic = WinCreateWindow( hwnd, WC_STATIC,
                                  NULL,
                                  SS_TEXT | WS_VISIBLE,
                                  10, 135, 120, 18,
                                  hwnd, HWND_TOP,
                                  CT_STATIC,
                                  NULL, NULL) ;

    hwndListbox = WinCreateWindow( hwnd, WC_LISTBOX,
                                    NULL,
                                    LS_NOADJUSTPOS | WS_VISIBLE,
                                    10, 10, 120, 120,
                                    hwnd, HWND_TOP,
                                    CT_LISTBOX,
                                    NULL, NULL) ;

    ...
}

```

The loading of all strings is performed with *WinLoadString()*, after assigning to the short identifier *i* the ID of the first text string:

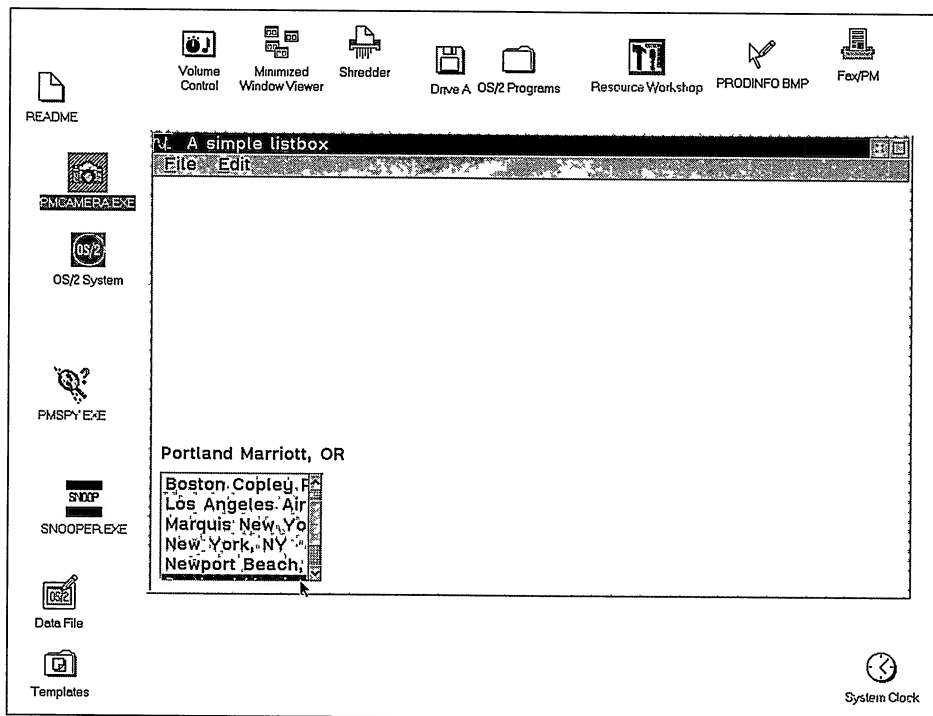
```

...
i = ST_MARRIOTT ;

while( WinLoadString( HAB( hwnd), NULLHANDLE, i +,
                    sizeof szString,
                    szString))
    WinSendMsg( hwndListbox, LM_INSERTITEM,
                MPFROMSHORT( LIT_SORTASCENDING),
                MPFROMP( szString)) ;
    break ;
...

```

The loop breaks automatically when there are no more text strings to be retrieved from the STRINGTABLE area—when the last sequential ID is reached. The insertion in the listbox is governed by sorting the text strings in ascending alphabetical order, thus generating an output (Figure 7.13) that is completely different from the sequence present in the resource file:



**Figure 7.13** A listbox with the style `LS_NOADJUSTPOS` is not restricted by the rules that set a value on the Y axis that is an integer multiple of the system font's height.

```

...
STRINGTABLE
{
    ...
    ST_MARRIOTT + 0, "Boston Copley Place, MA"
    ST_MARRIOTT + 1, "Newport Beach, CA"
    ST_MARRIOTT + 2, "Marquis New York, NY"
    ST_MARRIOTT + 3, "New York, NY"
    ST_MARRIOTT + 4, "San Francisco Marriott, CA"
    ST_MARRIOTT + 5, "Los Angeles Airport, CA"
    ST_MARRIOTT + 6, "Portland Marriott, OR"
    ST_MARRIOTT + 7, "Boca Raton, FL"
}
...

```

The selection of items listed in the listbox will cause the owner's window procedure to receive the message `WM_CONTROL`:

```

...
case WM_CONTROL:
    switch( SHORT1FROMMP( mp1))
    {
        case ID_LISTBOX:
            switch( SHORT2FROMMP( mp1))
            {
                case LN_SELECT:
                    {
                        CHAR szString[ 60] ;
                        short sPos ;

                        sPos = (SHORT)WinSendMessage( hwndListbox,
                                                    LM_QUERYSELECTION,
                                                    MPFROMSHORT(LIT_FIRST),
                                                    0L) ;

                        WinSendMessage( hwndListbox, LM_QUERYITEMTEXT,
                                        MPFROM2SHORT( sPos, sizeof szString),
                                        MPFROM( (PSZ)szString)) ;

                        WinSetWindowText( hwndStatic, szString) ;
                    }
                    break ;
                }
            }
        break ;
    }
    default:
        break ;
    }
    break ;
...

```

It is by intercepting the notification code LN\_SELECT that the application can identify and retrieve the selected item, then display it in the WC\_STATIC class window above the listbox. Naturally, the same logic extends to any other notification code. The application's source code is listed in Listing 7.5.



## Message Flow

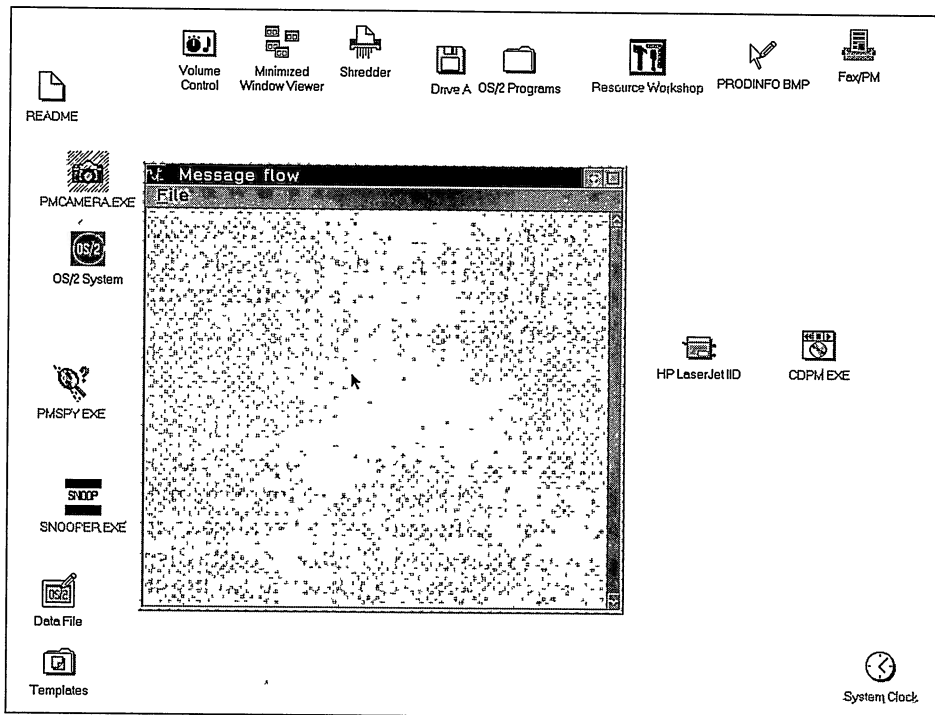
The windows of the class WC\_LISTBOX are an ideal tool to progressively list text strings in their order of arrival. The flag LIT\_END will in fact force the message LM\_INSERTITEM to append any new text string to the end of the list already in the listbox. This behavior is very useful if you need to capture and subsequently display the messages produced by a window when it is created. The utility program PMSPY.EXE furnished with the IBM Toolkit allows you to monitor the message flow of a window only after selecting it with the mouse. It is much more interesting to discover the precise sequence of messages fired by the *WinCreateStdWindow()* to the window procedure of the class to which the window belongs. Without resorting to the debugger, you can solve the problem, thanks to the services provided by the class WC\_LISTBOX. The program's structure is based on the following actions:

- Create a top-level window containing a window of class WC\_LISTBOX corresponding to its client window and with the same size of the client window.
- Create a window of another class registered by the application after a menu selection.
- Retrieve from the application's resource file the text strings that correspond to the numerical values of the messages received by the window procedure.
- Send the text strings retrieved from the resource file to the listbox contained in the application's main window.

The application registers two window classes. The application's main window has a menu bar to provide some interaction for the user. The second class window procedure has been designed to pass each message received to the main window equipped with a listbox overlapping the client.

## Creating a Listbox

The client area of the main window (Figure 7.14) is totally covered by the WC\_LISTBOX class window created in the message WM\_CREATE and sized through the message WM\_SIZE.



**Figure 7.14** A top-level window with a listbox totally overlapping the application's client window.

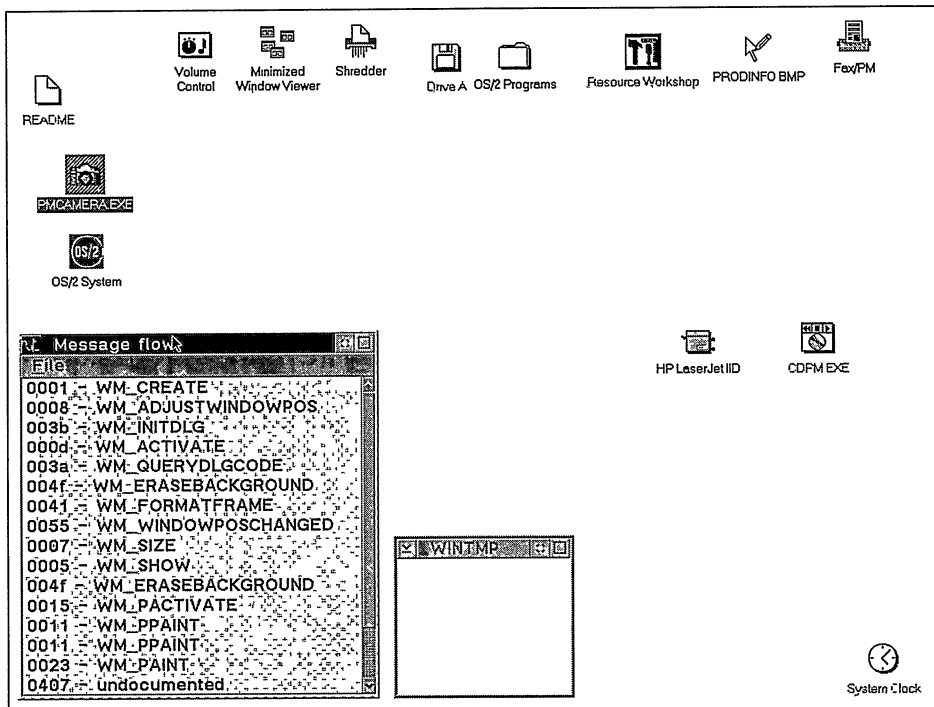


Thus, the client window simply plays the role of a pixel provider window for the listbox, and will receive the message `WM_COMMAND` each time the user performs a selection from the menu bar.

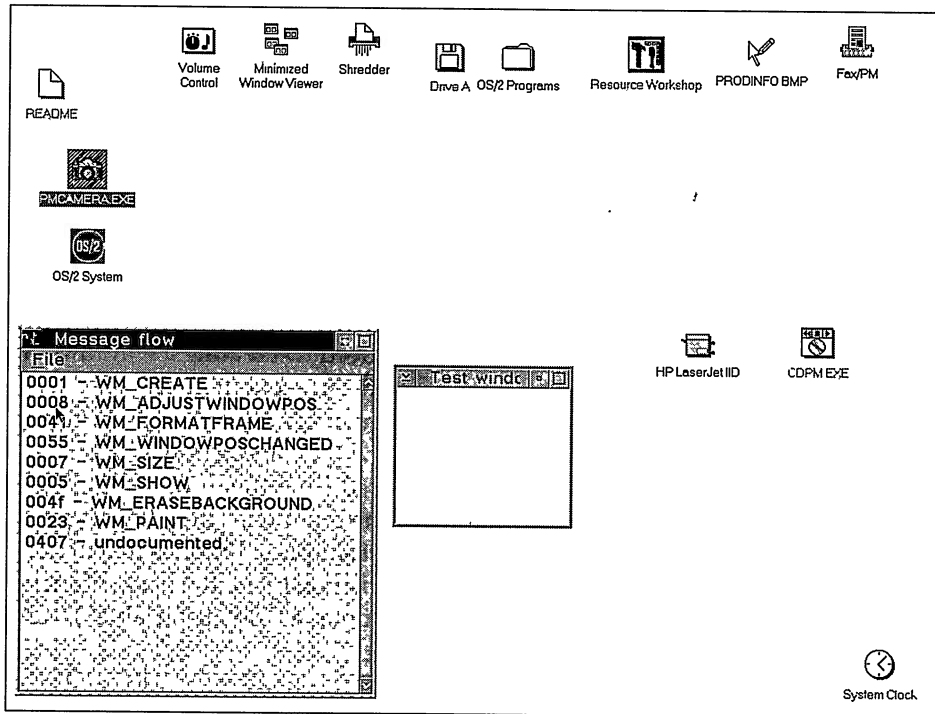
Two options are available in the File menu: Dialog and Window. Both of them are active immediately after loading the program, and, if selected, will cause the creation of a window, respectively, through the function `WinLoadDlg()` and `WinCreateStdWindow()`. These two alternatives allow you to further evaluate the two solutions already discussed in the previous chapters for creating a window. In Figure 7.15 you can see the program's output when the user selects the Dialog option when the window is created with `WinLoadDlg()`.

Much shorter is the sequence of messages produced by creating a window with the traditional `WinCreateStdWindow()` function, as you can see in Figure 7.16.

Both text windows share the same class, and thus rely on the same services of the class window procedure. This function must know the handle of the listbox in order to insert into the listbox all the text strings retrieved from the resource file. Since this window is created in the window procedure of the main window, there is no way to get it into the second window procedure. The most simple solution is that of declaring an identifier corresponding to the handle of the listbox *before* the `main()` function, so that it gets source file scope (in other words, it becomes a "global variable"). Actually,



**Figure 7.15** Message flow produced by loading a window template with the `WinLoadDlg()` function.



**Figure 7.16** Message flow generated by creating a window with *WinCreateStdWindow()*.

in the source code a slightly different logic, which can turn out to be useful even in other situations, is implemented.

The general rule is for the window procedure to be called indirectly only by the *WinDispatchMsg()* function in the message loop, by *WinSendMessage()*, or by several other API functions as a side effect (Chapter 3). In this case, though, we want to have access to the window procedure to communicate and store in the static storage class identifier the value of the listbox's handle. To achieve this result, it is necessary to have the name of the window procedure, a piece of information that can easily be retrieved through *WinQueryClassInfo()*:

```
...
// query second class window proc
WinQueryClassInfo( HAB( hwnd), szClassName2, &clsi );

// pass the hwndListBox handle
(*clsi.pfnWindowProc)(NULL, WM_SETUP, hwndListBox, 0L);
...
```

The transfer of execution to the window procedure of the second class takes place by using the name of the function, which in the C Language corresponds to a pointer

to the place in the code segment where the function's instructions start (it is pointer to a functions). The parameters passed to this indirect call are set according to the specific needs of the moment. There is no target window handle simply because we have not yet created any window of this class, while the message WM\_SETUP is defined in the following way prior to the *main()* function:

```
#define WM_SETUP WM_USER
```

In *mp1* you pass the handle of the listbox, while *mp2* is not used. The window procedure of the second class has in its *switch* block a case WM\_SETUP branch to intercept this application's enforced "message."

```
...
case WM_SETUP:
    hwndListBox = HwndFromMP( mp1 );
    return (MRESULT)TRUE ;

default:
    break ;
}

// send messages to the listbox
WinLoadString( HAB( hwnd ), NULLHANDLE, msg, sizeof( szMsg ), szMsg ) ;
WinSendMsg( hwndListBox, LM_INSERTITEM,
            MPFROMSHORT( LIT_END ),
            MPFROM( (PSZ)szMsg ) ) ;

return WinDefWindowProc( hwnd, msg, mp1, mp2 ) ;
}
...
```

The extraction of the values present in *mp1* is performed with the macro *HwndFromMP*, and the values are then stored in the identifier *hwndListBox* with static storage class. This message does not require the classic processing termination condition, but will have a forced exit from the window procedure through the keyword *return* in order to avoid executing the statements that are present before the default processing to which each message is submitted. In fact, at the end of the *switch* block, you find the call to the function *WinLoadString()* to retrieve the text string corresponding to the received message from the resource file, and then call *WinSendMsg()* to transfer the contents of the character array directly into the listbox.

It is this portion of code that is responsible for transferring the text strings, corresponding to the received messages, to the listbox in the main window. Calling a window procedure directly, as in this example (Listing 7.6), is a simple solution and, from our point of view, better than declaring an identifier with source file scope, which is an alternative that should be employed only when you want to write poor code.



## Owner-Drawn Listboxes

A listbox is an ordinary PM window belonging to a predefined class and whose window procedure resides internally in PM. It is not possible to know the structure of any of the predefined classes, but it is reasonable to think that there are intercepting conditions for the messages WM\_CREATE, WM\_PAINT, WM\_CLOSE, and many others. The rules that govern a listbox's output production must be found in the code fragment dealing with WM\_PAINT, and essentially pertaining to the display of some text strings extracted from among those that are present in the listbox's buffer.

The OS/2 2.1 API provides the flag LS\_OWNERDRAW to let you implement a listbox that delegates to its owner the task of handling all output details. This means that when the moment comes to display a text string, the listbox's window procedure will interrupt its operations, calling on the owner window to perform all needed tasks. The style LS\_OWNERDRAW indicates that it is the owner who will take the listbox's place in performing all output operations. The interaction that takes place between an *owner-drawn listbox* and the owner window is revealed by two distinct messages: WM\_MEASUREITEM and WM\_DRAWITEM. Figure 7.17 shows a scheme of the message flow that is generated by an owner-drawn listbox when it is processing the WM\_PAINT message (the same considerations also pertain to any other predefined window that delegates the drawing of their output).

### The WM\_MEASUREITEM Message

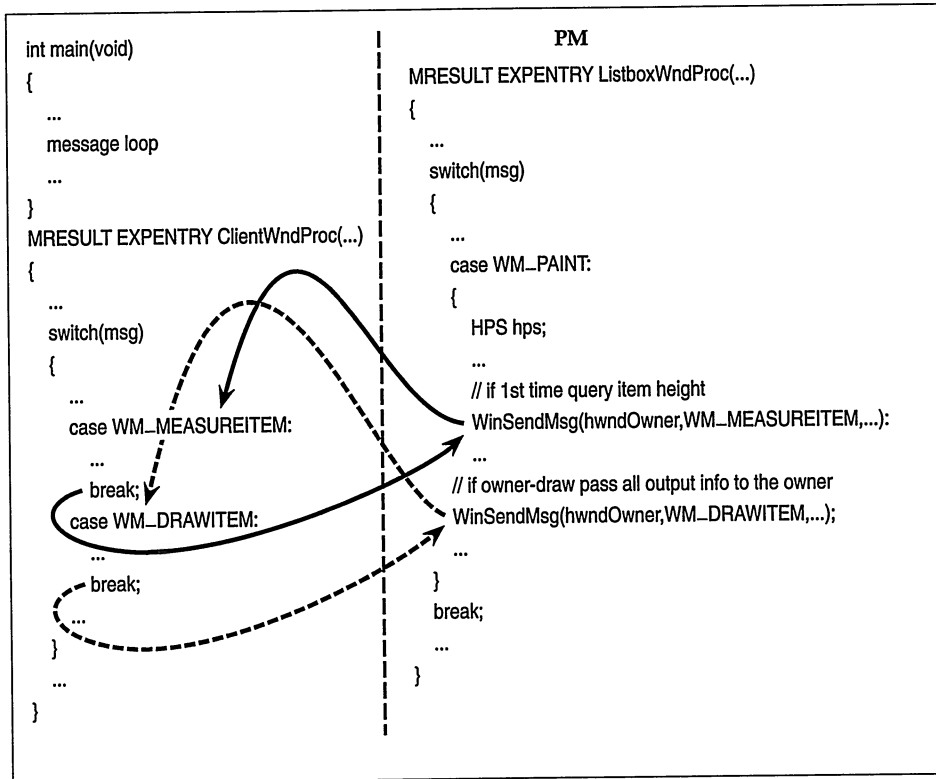
The listboxes of OS/2 2.1 will always show up with items all having the same height. This piece of data is generally computed by the listbox by evaluating the height of the system font. In the case of an owner drawn listbox, even this becomes the owner's duty. Consequently, the distance between two consecutive items can turn out to be bigger, smaller, or equal to the standard value, according to the application's specific needs. The window procedure of the owner window will catch the WM\_MEASUREITEM message and return a specific value without executing the standard processing provided by *WinDefWindowProc()*:

```
...
case WM_MEASUREITEM:
    return MRFROMLONG( cy ) ;
...
```

The value of *cy* corresponds to the height of the items in the window, and has been computed earlier in the program's code. The issuing of the message WM\_MEASUREITEM takes place only once for each listbox, just before the first item is displayed in the window.

### The WM\_DRAWITEM Message

The owner of an owner-drawn listbox will receive the message WM\_DRAWITEM when a listbox item needs to be displayed. This operation regards only the visible items, always a small number compared to all items present in the listbox.



**Figure 7.17** Scheme of message flow for an owner-drawn listbox.

		<i>Description</i>
WM_DRAWITEM	0x0036	
mp1	USHORT id	ID of the control.
mp2	POWNERITEM poi	Address of a OWNERITEM structure.
Return Value	BOOL fDrawn	Success or failure of the draw operation.

The first SHORT of mp1 contains the ID of the control that has issued the message (a menu or a listbox). In mp2 you find a pointer to a structure of type OWNERITEM, containing all information needed for delegating to the owner the drawing operations of the listbox's output.

```

typedef struct _OWNERITEM
{ // oi
  HWND hwnd ;
  HPS hps ;
  ULONG fsState ;
  ULONG fsAttribute ;
  ULONG fsStateOld ;
  ULONG fsAttributeOld ;
}

```

```

    RECTL rcItem ;
    LONG idItem ;
    ULONG hItem ;
} OWNERITEM ;

typedef OWNERITEM *POWNERITEM ;

```

Table 7.17 gives the meaning of each member in a OWNERITEM structure. The members of a OWNERITEM structure allow the owner window to draw inside the presentation space of a listbox. The actual control over output is delegated to another window, and you can therefore design code so as to insert almost any kind of object in a listbox, not being limited to text items only. In Figure 7.18 you can see the output of the program listed in Listing 7.7. The selection of the *File:Show documents* option will create eight child windows in a cascading layout. The lower-left corner accommodates a listbox, another child window of the application's client window, and thus a *sibling* to the frame windows of the documents.



In each of the listbox's items you can see a small item that looks like a rolled-up curtain and the name of a document as the reproduction of the text label present in the corresponding titlebars.

By selecting a text item in the listbox you can change the active document window. More interesting is the selection performed with the mouse by clicking over one of the icons. The corresponding document will not be selected; instead, the window disappears (in Figure 7.19, for instance, you can no longer see window number 0). Furthermore, the rolled-up curtain icon is now replaced by an image of a window with a pulled-down curtain.

**Table 7.17 Description of the Members of the OWNERITEM Structure used by PM for Displaying the Items in Owner-Drawn Listbox**

<i>Member</i>	<i>Description</i>
hwnd	Handle of the control that issued the message.
hps	Handle of the control's presentation space.
fsState	Indicates the selection state ( <i>highlighting</i> ).
fsAttribute	Contains the attribute flags.
fsStateOld	Indicates the preceding selection state ( <i>highlighting</i> ).
fsAttributeOld	Contains the preceding attribute flags.
rcItem	A RECTL structure describing the rectangle of the <i>listbox</i> that needs to be drawn.
idItem	Identifies the index (position) of the item to be redrawn inside the <i>listbox</i> , starting by counting from the origin.
hItem	Handle contained in the four bytes available for each item in a <i>listbox</i> .

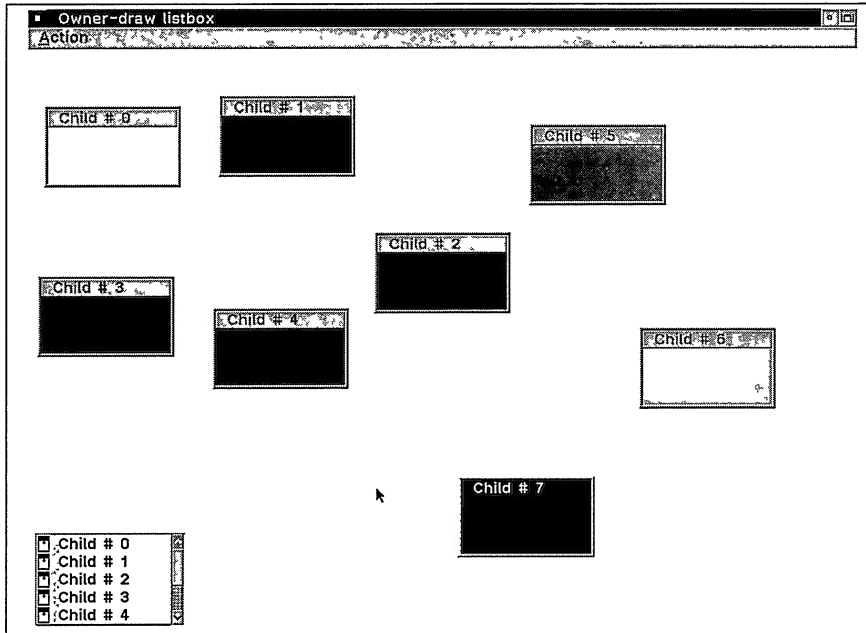


Figure 7.18 The listbox in the lower left-hand corner of the client area shows text and icons.

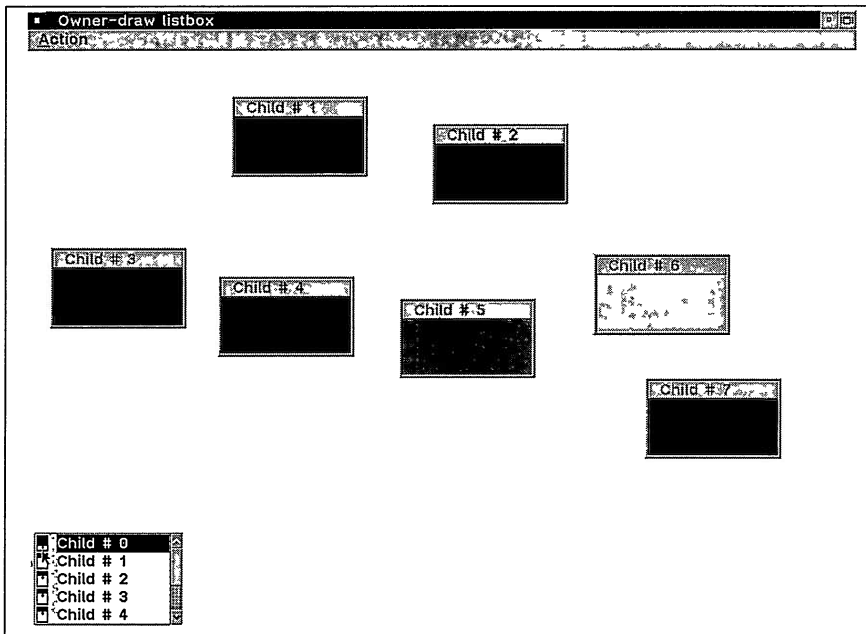
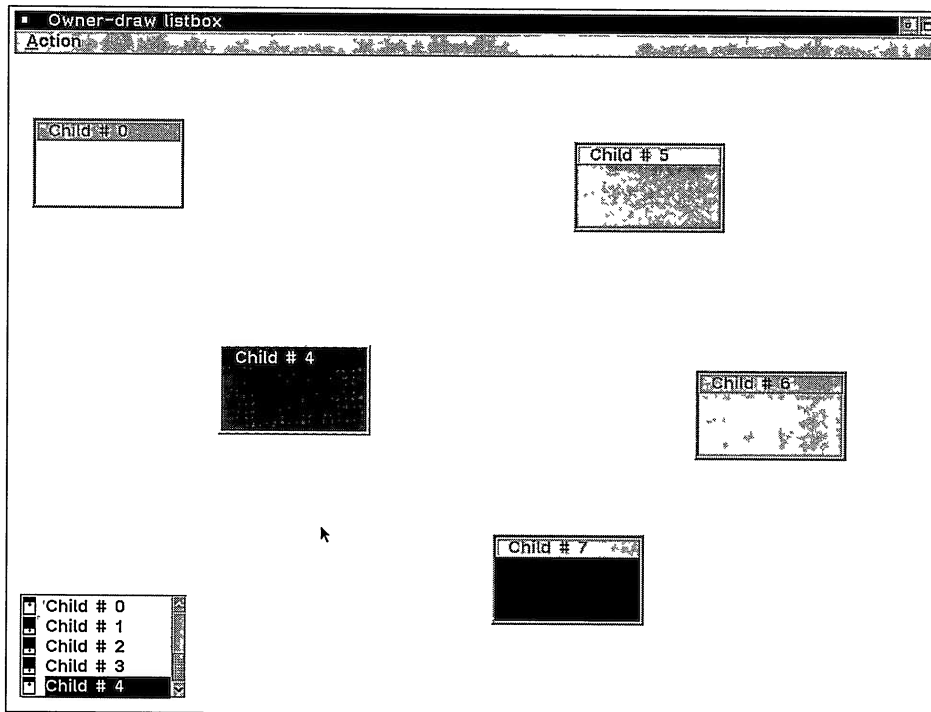


Figure 7.19 Selecting one of the listbox's items by clicking the corresponding icon (a rolled-up curtain) will hide the document from the user's view.



**Figure 7.20** The contents of an application-drawn listbox can be handled as you please, even implementing designs that go beyond the CUA specifications.

A subsequent click on the icon showing the pulled-down curtain will reveal the hidden document (Figure 7.20). As this is an ordinary listbox, you may not select the same item twice in a row, and thus you are not permitted to hide and then immediately reveal the same window. This behavior can be changed by setting the style `LS_MULTIPLESEL` or `LS_EXTENDEDSEL` when the window is created; however, when you activate any of these two styles, you will also allow the user to select more than one item at the same time, which is not good in a small program. Let's now examine the code of `ODLIST`.

## Creating a Listbox

All the windows of the application are children of the program's client window. Their creation takes place in the parent's window procedure. Before the listbox is actually created, the code will need to get the height of the system font by calling the function `GpiQueryFontMetrics()`, and then extract from a `FONTMETRICS` structure the value stored in the member `lMaxBaselineExt`. The size is then stored in a long variable with static storage class. This is the value that the application will return to the listbox whenever it receives the message `WM_MEASUREITEM`.



To do this, it is necessary to have a handle to a presentation space returned by the function *WinGetPS()*. It would have been better to get this information by acting directly on the presentation space of the listbox, rather than on the handle of the client area of the parent's window. This is not possible, because the height of the font must be known before the *WM\_MEASUREITEM* message is received, as this message is issued by the *WinCreateWindow()* regarding the listbox immediately after its execution. Furthermore, at that time no presentation space is yet available for the listbox (maybe this is a little bug in the API?).

The *hps* handle is also useful for loading the icons that will later be displayed in the listbox: This is done by calling *GpiLoadBitmap()*. The two bitmap handles are then stored in a static storage class array so that they can be used later whenever the message *WM\_DRAWITEM* is received.

```

...
case WM_CREATE:
{
    HPS hps ;
    FONTMETRICS fm ;
    PDATA pData ;
    // allocate the chunk of memory
    if( DosAllocMem( (PVOID)&pData, sizeof( DATA) * DOCNUMBER,
                    PAG_READ | PAG_WRITE | PAG_COMMIT))
        WinAlarm( HWND_DESKTOP, WA_ERROR) ;

    // store the pointer
    WinSetWindowPtr( hwnd, QWL_USER, pData) ;

    hps = WinGetPS( hwnd) ;
    GpiQueryFontMetrics( hps, sizeof( FONTMETRICS), &fm) ;
    cy = fm.lMaxBaselineExt ;
    hbm[ 1] = GpiLoadBitmap( hps, 0, 100, 0, 0) ;
    hbm[ 0] = GpiLoadBitmap( hps, 0, 101, 0, 0) ;
    WinReleasePS( hps) ;

    hwndListbox = WinCreateWindow( hwnd, WC_LISTBOX,
                                   NULL,
                                   LS_OWNERDRAW | WS_CLIPSIBLINGS,
                                   10, 10, 180, 120,
                                   hwnd, HWND_TOP,
                                   ID_LISTBOX,
                                   NULL, NULL) ;
}
break ;
...

```

The size of the listbox is computed so that it is an integer multiple of the system font's height. The setting of the *WS\_CLIPSIBLINGS* style has been added to avoid any possible problems caused by documents overlapping the listbox.

## Handling Information

The logic governing the application presented in Listing 7.7 causes the selection of a listbox item to activate the corresponding document. In order to implement this behavior the application adopts a rather unusual design, which for our purposes is interesting for discovering some of the peculiar quirks of listboxes. The windows involved by the application are:

- The main window belonging to a class of its own
- The document windows belonging to a different class
- The listbox window

The program has to allocate a memory area to store the information appearing in each document, as provided for by the header file ODLIST.H in the DATA structure:

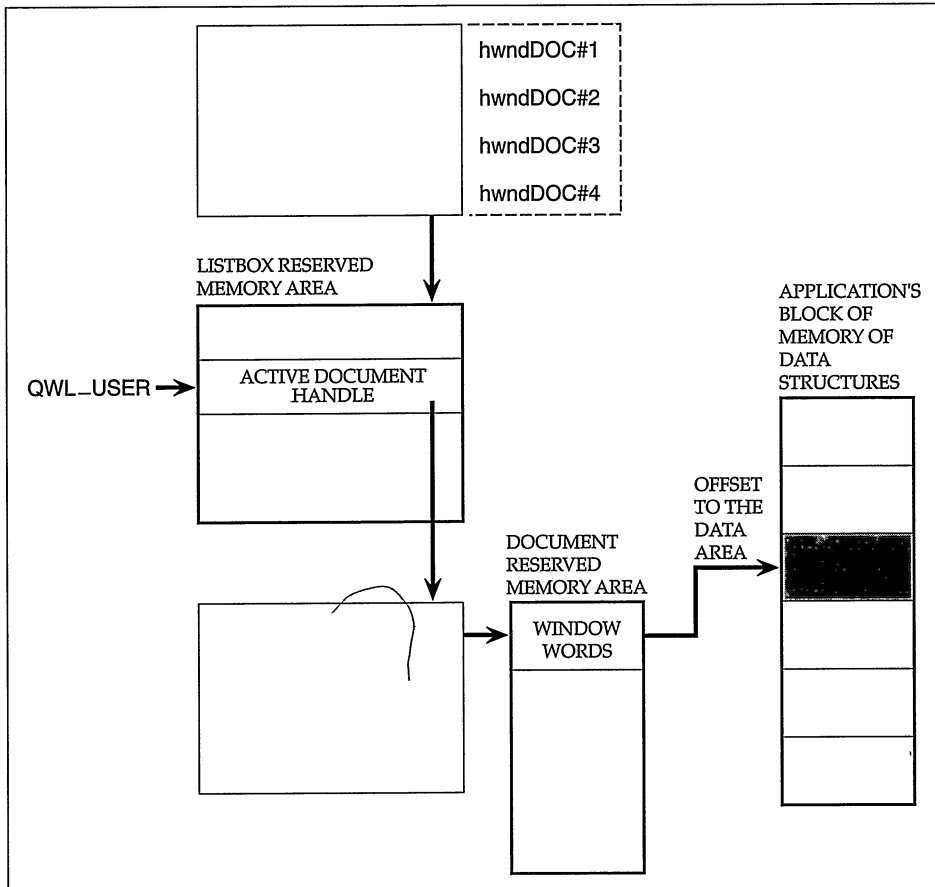
```
...
// define new data types
typedef struct _DATA
{
    BOOL fStatus ;
    COLOR clr ;
} DATA ;
typedef DATA * PDATA ;
...
```

The Boolean `fStatus` is used to define the presentation state of each single document. The color assigned to the window is encoded in `clr`. The application also allocates a memory block equal to the size of a DATA structure multiplied by the number of documents (`DOCNUMBER`), and stores the pointer obtained from `DosAllocMem()` in the window words of the main window. Each single document also has its own set of window words, where the window's position is stored with respect to the origin of the client window. The registration phase will consider all elements when preparing the needed space:

```
...
if( !WinRegisterClass( hab, szClassName,
                      ClientWndProc,
                      CS_SIZEREDRAW, sizeof( PDATA)))
    return FALSE ;

if( !WinRegisterClass( hab, szChildClass,
                      ChildWndProc,
                      CS_SIZEREDRAW, sizeof( LONG)))
    return FALSE ;
...
```

The program then allocates one memory block that will always be accessible to the main window's client by means of a pointer stored in the window words. Whenever necessary, the pointer is retrieved and updated to point to the correct block of data.



**Figure 7.21** Memory area used to store the information needed by the listbox to implement the behavior that allows the active document to be highlighted.

Furthermore, the listbox must always know the handle of the active document window. This information is stored directly in the reserved data area of the window belonging to the class `WC_LISTBOX`, by calling `WinSetWindowULong()`, and setting appropriately the `QWL_USER` flag. Each item in the listbox will take advantage of the four bytes it has available to store a handle to the client of the window to which the item is referring. Figure 7.21 summarizes this and delivers a schematic view of the logic followed in writing this application.

## Drawing an Item

When the option *Show Documents* is selected, it will display the documents on the screen together with the listbox containing the text strings corresponding to the titles shown in the titlebars. This is the first place in the program where you need to draw

the contents of the listbox. The application will receive the message WM\_DRAWITEM in the window procedure of the owner, with mp2 containing the address of an OWNERITEM structure. As we have already seen in Table 7.17, the OWNERITEM structure does not contain the text that has to be displayed in the listbox. To get to that, you must issue the message LM\_QUERYITEMTEXT and store the text string in an array of characters. Some of the information contained in the RECTL member of OWNERITEM are used as starting points for defining the item's area to be occupied by the icon previously loaded, namely in the message WM\_CREATE. The new rectangle is stored in a second RECTL structure, used when you need to display the icon. Before doing this it is necessary to evaluate if the current position of the mouse pointer corresponds to the area designated for containing the icon. To discover the coordinates of the mouse, call *WinQueryPointerPos()* rather than *WinQueryMsgPos()*, because this last function is applicable only for posted messages, while the messages actually being considered here are all sent messages,

```
#define INCL_WINPOINTERS
BOOL WinQueryPointerPos( HWND hwndDesktop, PPOINTL pptl ) ;
```

<i>Parameter</i>	<i>Description</i>
hwndDesktop	Handle to the desktop window
pptl	Address of a POINTL structure
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The parameters to this function are the desktop's handle or the define HWND\_DESKTOP and a pointer to a structure of type POINTL with the mouse's current position expressed in screen coordinates. At this moment the mouse pointer is over the drop-down File menu and can in no way affect the listbox's output. To see if the mouse's coordinates correspond to the position of the listbox, it is not sufficient to simply convert them via *WinMapWindowPoints()*, because the converted point might fall outside the area occupied by the listbox on the screen (which is precisely what happens here). The subsequent comparison between the returned handle and the listbox handle resolves the problem. Only if there is a match will the program proceed in determining if the user clicked the left mouse button over the icon.

Other operations must be performed before actually executing the comparison. Through the *idItem* member of the OWNERITEM structure, the member that identifies the position of the item to be displayed, the code issues the message LM\_QUERYITEMHANDLE to the listbox in order to get a handle to the client area of the corresponding document window. Once the handle is available, you can access the offset present in the window words and add it to the base pointer.

```
...
case WM_DRAWITEM:
{
    RECTL rc1 ;
    OWNERITEM * pitem ;
    LONG clrForeground, clrBackGround ;
```

```

POINTL pt1 = { 0, 0 } ;
CHAR szbuf[ 80 ] ;
BITMAPINFOHEADER bmp ;
FONTMETRICS fm ;
HWND hwndDoc, hwndCheck ;
PDATA pData ;
LONG lOffset ;

// pointer to the OWNERITEM struct
pitem = (POWNERITEM)PVOIDFROMMP( mp2 ) ;

// determine the text
WinSendMessage( pitem -> hwnd, LM_QUERYITEMTEXT,
                MPFROM2SHORT( pitem -> idItem, sizeof szbuf),
                (MPARAM)(PSZ)szbuf) ;

// icon's rectangle
rc.xLeft = pitem -> rcItem.xLeft ;
rc.yBottom = pitem -> rcItem.yBottom ;

GpiQueryBitmapParameters( hbm[ 0 ], &bmp ) ;
rc.xRight = cx = rc.xLeft + bmp.cx ;
rc.yTop = rc.yBottom + cy ;

// check the pointer pos
WinQueryPointerPos( HWND_DESKTOP, &pt1 ) ;
hwndCheck = WinWindowFromPoint( HWND_DESKTOP, &pt1, TRUE ) ;

// get the doc hwnd
hwndDoc = (HWND)WinSendMessage( hwndListBox, LM_QUERYITEMHANDLE,
                                MPFROMSHORT( pitem -> idItem ), 0L ) ;

// get the data pointer
pData = (PDATA)WinQueryWindowPtr( hwnd, QWL_USER ) ;
lOffset = (LONG)WinQueryWindowULong( hwndDoc, QWL_USER ) ;
pData += lOffset ;
...
}
...

```

Only if the handle of the window underlying the mouse's position corresponds to the listbox will the code proceed by converting the point of the cursor's hot spot into window coordinates, and then check if that is a location within the area occupied by the icon. If it is, the application modifies the value of the `fStatus` member of the data structure created specifically for the application (DATA) and then changes the visibility status of the window.

```

...
// are you clicking on the listbox?
if( hwndCheck == hwndListBox )
{
    WinMapWindowPoints( HWND_DESKTOP, hwndListBox, &pt1, 1L ) ;
    // is the mouse on the icon?

```

```

if( ptl.x > 0 && ptl.x <= ( cx + OFFSET) && pitem -> fsState)
{
    WinAlarm( HWND_DESKTOP, WA_NOTE) ;
    pData -> fStatus = ! pData -> fStatus ;
    // show/hide the doc window
    WinSetWindowPos( PAPA( hwndDoc), HWND_TOP,
                    0L, 0L, 0L, 0L,
                    ( pData -> fStatus) ?
                    (SWP_SHOW | SWP_ACTIVATE) : SWP_HIDE) ;
}
}
...

```

The only thing left to be done is to display the appropriate icon with `WinDrawBitmap()`, being careful to position it in the area previously identified, and finally to display the corresponding text with `WinDrawText()`.

```

...
// no highlight on the selected item
if( pitem -> fsState == FALSE )
{
    clrForeground = CLR_BLACK ;
    clrBackGround = CLR_WHITE ;
}
else
{
    clrForeground = CLR_WHITE ;
    clrBackGround = CLR_BLACK ;
}

// draw the icon
WinDrawBitmap( pitem -> hps,
               ( pData -> fStatus) ? hbm[ 0] : hbm[ 1],
               NULL,
               (PPOINTL)&rc,
               CLR_BLACK, CLR_WHITE,
               DBM_STRETCH) ;

// determining the text rectangle
rc1.xLeft = pitem -> rc1Item.xLeft + bmp.cx + OFFSET ;
rc1.xRight = pitem -> rc1Item.xRight ;
rc1.yTop = pitem -> rc1Item.yTop ;
rc1.yBottom = pitem -> rc1Item.yBottom ;

// writing text
WinDrawText( pitem -> hps,
             -1,
             szbuf,
             &rc1,
             clrForeground, clrBackGround,
             DT_LEFT | DT_VCENTER | DT_ERASERECT) ;

...

```

The final phase of processing the WM\_DRAWITEM message deals with toggling the members fsState and fsStateOld of the OWNERITEM structure, for the ensuing painting operations.

### Some Considerations

The example illustrated in Listing 7.7 shows how it is possible to position any kind of object in the presentation space of a listbox, and thereby implement new kinds of behavior of this window. This is a simple but efficient example of going beyond the CUA specifications of OS/2 Presentation Manager.

---

## The Class WC\_ENTRYFIELD

One of the most common operations in a generic application is requesting information directly from the user without going through controlled input structures like menus. The user's name when installing a program, and the date of an invoice, are typical examples. To make it easier to perform this kind of operation, the PM API provides the designer with the class WC\_ENTRYFIELD, which specializes in handling keyboard input of numeric or alphanumeric data. Almost all dialog windows that appear in PM applications resort to WC\_ENTRYFIELD class windows, and relieve the designer of the task of handling WM\_CHAR messages and controlling the cursor movements.

### The ES\_ Styles

Table 7.18 lists all ES\_ styles that affect the look and behavior of the windows belonging to this class.

The combination of ES\_MARGIN and ES\_READONLY styles will produce a window that can display any text preset by the application; however it will not allow the user to change that text or type any new text. This solution is often used to offset the lack of a frame around windows belonging to the class SS\_STATIC. An even simpler solution is to set the style WS\_DISABLED in addition to those typical of a WC\_ENTRYFIELD class window. In such a case you disable any input activity, and transform an *entryfield* into a *static* from the user perspective.

**Table 7.18 The ES\_ Styles of the Class WC\_ENTRYFIELD**

<i>Style</i>	<i>Value</i>	<i>Description</i>
ES_LEFT	0x00000000L	The text is left-aligned in the window.
ES_CENTER	0x00000001L	The text is centered in the window.
ES_RIGHT	0x00000002L	The text is right-aligned in the window.

(continued)

Table 7.18 (Continued)

<i>Style</i>	<i>Value</i>	<i>Description</i>
ES_AUTOSCROLL	0x0000004L	The window performs horizontal scrolling automatically whenever the user types a number of characters that is greater than those that can be displayed in the window.
ES_MARGIN	0x0000008L	The perimeter of the window is surrounded by a thin border.
ES_AUTOTAB	0x0000010L	Automatically transfers focus to the next <i>control</i> in the TAB chain whenever the window contains a number of characters equal to the maximum allowed number.
ES_READONLY	0x0000020L	You are not allowed to perform any kind of input in the window; it will only display information.
ES_COMMAND	0x0000040L	This value is useful only for an <i>entryfield</i> in a Help panel.
ES_UNREADABLE	0x0000080L	Displays any character typed at the keyboard as an asterisks.
ES_AUTOSIZE	0x0000200L	Assigns to the windows an appropriate size for containing the text indicated during the window's creation.
ES_ANY	0x0000000L	The text inserted into the <i>entryfield</i> can be any combination of characters of one or two bytes.
ES_SBCS	0x0010000L	The window accepts only single byte characters.
ES_DBCS	0x0020000L	The window accepts only double byte characters.
ES_MIXED	0x0030000L	Like ES_ANY, but allows the conversion of DBCS ASCII characters into equivalent EBCDIC characters without causing <i>overflow</i> errors in the application.

## The EM\_ Messages

The best way to insert text into an entryfield is to specify it when the window is created. Later, this task can be performed with *WinSetWindowText()* by specifying the appropriate handle. The retrieval of the contents of an entryfield becomes accessible to the application by calling *WinQueryWindowText()*. Table 7.19 lists all EM\_ that are specific to the windows belonging to the class WC\_ENTRYFIELD.

The actions performed by the EM\_ messages essentially pertain to the handling of the window's output. The designer can change the first character displayed in the



**Table 7.19** The EM\_ Messages of the Class WC\_ENTRYFIELD

<i>Message</i>	<i>Value</i>	<i>Description</i>
EM_QUERYCHANGED	0x0140	Issued to determine whether or not the contents of the <i>entryfield</i> have been changed with respect to an earlier query with EM_QUERYCHANGED or WM_QUERYWINDOWPARAMS.
EM_QUERYSEL	0x0141	Returns the number of characters from the first selected character.
EM_SETSEL	0x0142	Set the selection of text by specifying the first and the last characters to be selected.
EM_SETTEXTLIMIT	0x0143	Defines the maximum number of characters that can be typed into an <i>entryfield</i> .
EM_CUT	0x0144	Cuts out the selected text and transfers it into a buffer.
EM_COPY	0x0145	Copies the selected text into a buffer.
EM_CLEAR	0x0146	Deletes the selected text.
EM_PASTE	0x0147	Pastes the previously copied or deleted text from the buffer to the current cursor position.
EM_QUERYFIRSTCHAR	0x0148	Returns the position of the first visible character in the <i>entryfield</i> .
EM_SETFIRSTCHAR	0x0149	Sets the first visible character in the <i>entryfield</i> .
EM_QUERYREADONLY	0x014a	Returns the status of the read-only attribute.
EM_SETREADONLY	0x014b	Activates/Deactivates the read only flag.
EM_SETINSERTMODE	0x014c	Activates/Deactivates the insert/overwrite mode.

window (the text might scroll horizontally by setting the ES\_AUTOSCROLL flag), define the set of selected characters, and set the read-only attribute. More interesting is the interaction with the Clipboard through messages for copying, clearing, and pasting the text.

## Notification Codes

A window of the class WC\_ENTRYFIELD will notify its owner of the actions performed by the user by including the notification codes listed in Table 7.20 as part of the WM\_CONTROL message.

To know when the contents of an entryfield change, you have to intercept EN\_CHANGE. This notification code will also appear if the user presses the backspace key to correct previously written characters.

**Table 7.20 The Notification Codes of the Class WC\_ENTRYFIELD**

<i>Notification Codes</i>	<i>Value</i>	<i>Description</i>
EN_SETFOCUS	0x0001	Focus has been transferred to an <i>entryfield</i> window.
EN_KILLFOCUS	0x0002	Focus has been transferred from an <i>entryfield</i> window.
EN_CHANGE	0x0004	There has been a change in the <i>entryfield</i> window.
EN_SCROLL	0x0008	The number of characters typed in has caused a horizontal <i>scrolling</i> of the text.
EN_MEMERROR	0x0010	Informs that the <i>entryfield</i> was not able to allocate a memory area large enough to contain the number of characters requested by the message EM_SETTEXTLIMIT.
EN_OVERFLOW	0x0020	Sent to notify that the maximum limit of characters has been reached in the <i>entryfield</i> .
EN_INSERTMODETOGGLE	0x0040	Input handling has changed by toggling from insert mode to overwrite mode, or vice versa.

## The ENTRYFDATA Structure

The class WC\_ENTRYFIELD has a data structure called ENTRYFDATA which, provided it is filled with appropriate values, allows you to customize the behavior of an entryfield from the very moment it is created. In version 2.1 of OS/2, the ENTRYFDATA contains four USHORT members:

```
#define INCL_WINENTRYFIELDS
typedef struct _ENTRYFDATA
{ // efd
    USHORT cb ;
    USHORT cchEditLimit ;
    USHORT ichMinSel ;
    USHORT ichMaxSel ;
} ENTRYFDATA ;

typedef ENTRYFDATA *PENTRYFDATA ;
```

The member `cchEditLimit` carries out the same action as the message `SETTEXTLIMIT` and sets the maximum number of characters that can be typed into the window. The third and fourth members, `ichMinSel` and `ichMaxSel`, define the lower and the upper limit of the selected text. In future versions of PM, this structure will allow the establishment of an input format by defining the number, the type, and the layout of data items that the user can input through the keyboard. Listing 7.8 shows a window of class WC\_ENTRYFIELD at work.



## The Class WC\_COMBOBOX

The representatives of this class constitute a kind of compromise between a listbox window and an entryfield. The very name suggests a combination of simpler elements, just like the classes WC\_LISTBOX and WC\_ENTRYFIELD. The principal purpose of this class is to provide the designer with a tool that has the same degree of flexibility as a listbox, and the size of a simple entryfield. In fact, a combobox shows up on the screen like an ordinary entryfield, with the addition of an a downward pointing arrow icon to its right side. Clicking the left mouse button on the icon will cause a listbox to be displayed beneath the entryfield. The item selected from the listbox is displayed automatically in the entryfield part of the combobox, causing the listbox to disappear.

### The CBS\_ Styles

The amount of aesthetic and functional customization of a combobox is rather limited. The alternatives are shown in Table 7.21.

The solution CBS\_DROPDOWNLIST is a very flexible and useful input tool. The user is given the choice of selecting an item from the listbox part without modifying the entryfield part. Such a combobox can be used when you want to give the user the possibility of choosing among several predefined values, or among values that might possibly grow in number through a mechanism that does not depend on the user input. Also, CBS\_DROPDOWN is used quite often, while CBS\_SIMPLE is seldom needed.

**Table 7.21 The Styles of the Class WC\_COMBOBOX**

<i>Style</i>	<i>Value</i>	<i>Description</i>
CBS_SIMPLE	0x0001L	Creates a combination of an <i>entryfield</i> and a <i>listbox</i> and displays them on the screen simultaneously: The user can both select an item from the <i>listbox</i> as well as type some text directly in the <i>entryfield</i> .
CBS_DROPDOWN	0x0002L	Only the <i>entryfield</i> will be displayed on the screen, together with a downward pointing arrow icon to its right, which will display the <i>listbox</i> when selected: The user can both select an item from the <i>listbox</i> as well as type some text directly in the <i>entryfield</i> .
CBS_DROPDOWNLIST	0x0004L	Only the <i>entryfield</i> will be displayed on the screen, together with a downward pointing arrow icon to its right, which will display the <i>listbox</i> when selected: The user is only allowed to select an item from the <i>listbox</i> , and cannot type any text in the <i>entryfield</i> .

**Table 7.22 The Messages of the Class WC\_COMBOBOX**

<i>Message</i>	<i>Value</i>	<i>Description</i>
CBM_SHOWLIST	0x0170L	Forces the <i>listbox</i> to be displayed or removes it from the screen: It is equivalent to clicking the left mouse button on the arrow icon.
CBM_HILITE	0x0171L	Forces the status of <i>highlighting</i> over the combobox's button.
CBM_ISLISTSHOWING	0x0172L	Returns the status of the <i>listbox</i> : displayed or hidden.

### *The CBM\_ Messages*

The designers of PM have confined to the bare minimum the number of messages, styles, and notification codes that are specific to comboboxes, and allowed that equivalent elements of listboxes and entryfields could be used just as effectively to perform specific operations. Thus, they avoided a redundancy in the API, which does not happen in MS Windows. There are only three messages of the class WC\_COMBOBOX (Table 7.22).

All messages of the class WC\_LISTBOX work correctly in the listbox portion of a combobox. For instance, to insert a text item in this window, you can use the message LM\_INSERTITEM and specify the appropriate handle: The same rules extend to any other specific operation. Instead, to insert text in or retrieve text from the entryfield portion, you can use the functions *WinSetWindowText()* and *WinQueryWindowText()*.

### *The Notification Codes*

A combobox notifies its owner window about the operations performed in response to the user's actions by sending the notification codes listed in Table 7.23 as an integral part of the message WM\_CONTROL.

As you can see, it is a kind of combination of the notification codes of the classes WC\_LISTBOX and WC\_ENTRYFIELD.

### *Using a Combobox*

To demonstrate the characteristics of a combobox, let's implement a simple application that allows you to change the background color of the client window according to the user's selection inside the listbox portion (Listing 7.9). The logic is based on the interception of the window's notification codes and on the consequent painting activity. The code also intercepts the message WM\_WINDOWPOSCHANGED to determine if the resizing of the application's main window will interfere with the combobox's output (Figure 7.22).

Table 7.23 The Notification Codes of the Class WC\_COMBOBOX

Notification Code	Value	Description
CBN_EFCHANGE	1	The text in the <i>entryfield</i> has been changed.
CBN_EFSCROLL	2	Horizontal <i>scrolling</i> took place in the <i>entryfield</i> .
CBN_MEMERROR	3	The combobox was not able to allocate an area large enough to cater to the needs of the <i>entryfield</i> and/or the <i>listbox</i> .
CBN_LBSELECT	4	A <i>listbox</i> item was selected.
CBN_LBSCROLL	5	Vertical <i>scrolling</i> took place in the <i>listbox</i> .
CBN_SHOWLIST	6	The <i>listbox</i> was displayed/hidden.
CBN_ENTER	7	Signals a double-click with the left mouse button or the user's pressing the Enter key.

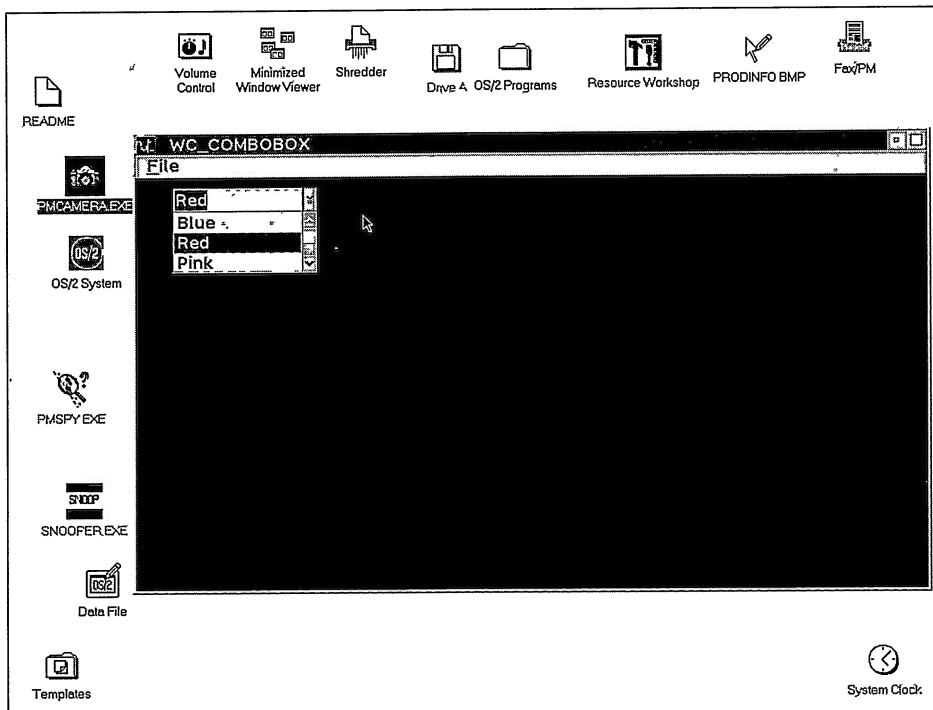


Figure 7.22 The combobox lists the names of the colors that can be selected for the client window.

When the CBN\_LBSELECT notification code is intercepted, the whole of the client area is invalidated with *WinInvalidateRect()*. However, updating is not enforced because it would produce undesired side effects in the listbox portion. Furthermore, when the main window is created, it is necessary to set the flag WS\_CLIPCHILDREN—otherwise, the pixels of the listbox would not be controlled by the application’s client window, and thus the image of the listbox would persist on the screen.

---

## The Class WC\_MLE

The class WC\_MLE (*multiple line entryfield*) was added to PM’s API with the introduction of version 1.2. There is also a special header file, PMMLE.H, that contains all the defines specific for this class. An *mle* is, in practice, an entryfield that spans several lines—an ideal tool for creating a word processor and thus relieving the designer of a great deal of trouble.

### *The Styles of the Class WC\_MLE*

Table 7.24 describes the specific styles of the WC\_MLE class. Customization is possible when creating the window, including the addition of scrollbars (both vertical and horizontal), the presence of a thin border around the four sides, the read-only attribute, and the ability to word wrap words longer than the remaining space available on a line.

**Table 7.24 The Styles of the Class WC\_MLE**

<i>Style</i>	<i>Value</i>	<i>Description</i>
MLS_WORDWRAP	0x00000001L	Wraps words longer than the remaining space available on a line to the next line.
MLS_BORDER	0x00000002L	Draws a thin border around the perimeter of the <i>mle</i> window.
MLS_VSCROLL	0x00000004L	Adds a vertical <i>scrollbar</i> to the right of the <i>mle</i> window.
MLS_HSCROLL	0x00000008L	Adds a horizontal <i>scrollbar</i> to the lower side of the <i>mle</i> window.
MLS_READONLY	0x00000010L	Disallows typing any text into the <i>mle</i> window; the user is allowed to only view the information.
MLS_IGNORETAB	0x00000020L	Does not consider the TAB key as a true tab character.
MLS_DISABLEUNDO	0x00000040L	The window will not support any kind of undo actions.

---

To define the number of rows that characterize an *mle* you have only to specify the vertical size of the window in parent units. The number of text lines is computed automatically by the *mle* itself, on the basis of the font being used. Therefore, it is possible that the height of an *mle* may not be an integer multiple of the system font, though this should not be a problem.

Both scrollbars will obtain their pixels directly from the *mle*, that thus is their owner. The scrolling of text, and the related movements of the slider, are controlled and handled automatically by the *mle* in cooperation with the scrollbars, and do not require any kind of provision on part of the programmer. The style `MLS_WORDWRAP` should be avoided because it slows down all text input operations in the *mle*.

## The MLM\_ Messages

The set of messages specific to the `WC_MLE` class is large, although it does not include all the functionality that you might expect (Table 7.25), as we will see later.

The list of messages of the class `WC_MLE` is quite extensive, and allows you to perform a number of operations on the windows of this class. The only deficiency is that of some message that would allow you to act directly upon the vertical scrollbar in order to reach a precise position in the text; something along the lines of “go to line xxx.”

**Table 7.25** The Messages of the Class `WC_MLE`

<i>Message</i>	<i>Value</i>	<i>Description</i>
<code>MLM_SETTEXTLIMIT</code>	0x01b0	Defines the maximum size in terms of number of characters that can be inserted in an <i>mle</i> . The amount of text that can be handled by an <i>mle</i> is theoretically unlimited, but practically restricted by the amount of memory available. By indicating the value of -1 you instruct the <i>mle</i> to continue accepting new text indefinitely, until system resources are exceeded.
<code>MLM_QUERYTEXTLIMIT</code>	0x01b1	Returns the size of the text that is allowed to be inserted in an <i>mle</i> : A return value of -1 indicates an unlimited number of characters.
<code>MLM_SETFORMATRECT</code>	0x01b2	Sets the size of the <i>editing</i> area inside the <i>mle</i> by defining horizontal and vertical limits that are different from the actual size of the window on the screen.

(continued)

**Table 7.25 (Continued)**

<i>Message</i>	<i>Value</i>	<i>Description</i>
MLM_QUERYFORMATRECT	0x01b3	Returns the size of the <i>editing</i> area inside the <i>mle</i> on the basis of an earlier definition through the sending of the message MLM_SETFORMATRECT.
MLM_SETWRAP	0x01b4	Toggles on <i>word wrapping</i> of text inside the editing area.
MLM_QUERYWRAP	0x01b5	Returns the status of the <i>word-wrapping</i> toggle.
MLM_SETTABSTOP	0x01b6	Defines the width of the tab character inside the <i>mle</i> ; the amount is expressed in number of pixels.
MLM_QUERYTABSTOP	0x01b7	Returns the width of the tab character inside the <i>mle</i> ; the amount is expressed in number of pixels.
MLM_SETREADONLY	0x01b8	Sets the read-only attribute in the <i>mle</i> .
MLM_QUERYREADONLY	0x01b9	Returns the status of the read-only attribute of the <i>mle</i> .
MLM_QUERYCHANGED	0x01ba	Returns a flag indicating whether the text contained in the <i>mle</i> has been changed since the last request for the same information.
MLM_SETCHANGED	0x01bb	Sets a flag that informs the application that the text contained in the <i>mle</i> has been changed.
MLM_QUERYLINECOUNT	0x01bc	Returns the number of text lines present in the <i>mle</i> .
MLM_CHARFROMLINE	0x01bd	Returns the number of characters that separate the first symbol on the line specified in the message from the start of the text in the <i>mle</i> .
MLM_LINEFROMCHAR	0x01be	Returns the number of the line in the <i>mle</i> that contains the specified character.
MLM_QUERYLINELENGTH	0x01bf	Returns the length of a line of text in the <i>mle</i> , starting the count from the indicated character or from the current cursor position.
MLM_QUERYTEXTLENGTH	0x01c0	Returns the overall number of characters present in the <i>mle</i> window.

(continued)





**Table 7.25 (Continued)**

<i>Message</i>	<i>Value</i>	<i>Description</i>
MLM_SETSEL	0x01ca	Defines the area of selected text in a <i>mle</i> by setting an initial and final position. If the initial position is greater than the final position, then the cursor will be positioned to the left of the selected text; otherwise, the cursor will always be at the right of the selected text.
MLM_QUERYSEL	0x01cb	Returns the position, in terms of characters from the start of text, of the first character of the selected text.
MLM_QUERYSELTEXT	0x01cc	Copies the selected text directly to a buffer specified in the message.
MLM_QUERYUNDO	0x01cd	Determines if it is possible to perform a specific undo operation in a <i>mle</i> : The <i>high word</i> of the return value contains a number corresponding to the MLM_ messages that can be undone, while the <i>low word</i> contains TRUE if undo is allowed or FALSE if the message has just been undone.
MLM_UNDO	0x01ce	Performs the undo operation on the contents of the <i>mle</i> .
MLM_RESETUNDO	0x01cf	Resets the undo flag in a <i>mle</i> ; the flag is set automatically whenever an undoable operation has been performed.
MLM_QUERYFONT	0x01d0	Returns in a FATTRS structure information regarding the font used in the <i>mle</i> .
MLM_SETFONT	0x01d1	Defines the font to be used in the <i>mle</i> by specifying information directly in a FATTRS structure.
MLM_SETTEXTCOLOR	0x01d2	Defines the text color.
MLM_QUERYTEXTCOLOR	0x01d3	Returns a numeric code representing the text color.
MLM_SETBACKCOLOR	0x01d4	Defines the background color of a <i>mle</i> window.
MLM_QUERYBACKCOLOR	0x01d5	Returns a numeric code representing the background color of an <i>mle</i> window.

(continued)

Table 7.25 (Continued)

<i>Message</i>	<i>Value</i>	<i>Description</i>
MLM_QUERYFIRSTCHAR	0x01d6	Returns the distance in characters of the first visible character from the start of text in the <i>mle</i> .
MLM_SETFIRSTCHAR	0x01d7	Defines which should be the first visible character in an <i>mle</i> window, notwithstanding the current position in the text.
MLM_CUT	0x01d8	Cuts out from an <i>mle</i> the selected text and transfers it to the buffer area so it is available for future <i>pasting</i> operations.
MLM_COPY	0x01d9	Copies the selected text in the <i>mle</i> to the buffer area, so it is available for future pasting operations.
MLM_PASTE	0x01da	Pastes the text currently in the buffer area to the current cursor position in the <i>mle</i> .
MLM_CLEAR	0x01db	Deletes the selected text from the <i>mle</i> .
MLM_ENABLEREFRESH	0x01dc	Enables <i>refresh</i> of output in the <i>mle</i> .
MLM_DISABLEREFRESH	0x01dd	Disables <i>refresh</i> of output in the <i>mle</i> , also preventing any kind of input operation (by mouse or keyboard).
MLM_SEARCH	0x01de	Searches for and replaces a piece of text in the <i>mle</i> .
MLM_QUERYIMPORTEXP	0x01df	Returns the address and the size of the buffer area used for <i>import/export</i> operations to/from the <i>mle</i> .

### The MLN\_ Notification Codes

The number of notification codes of the class WC\_MLE is considerable (Table 7.26), as you might have inferred from the number of MLM\_ messages. All notification codes will reach the owner's window procedure by means of the message WM\_CONTROL.

### Data Structures of the Class WC\_MLE

The complex nature of the windows of the class WC\_MLE imply the presence of several data structures in order to implement complex operations, like searching and replacing pieces of text. In addition, to the MLEMARGSTRUCT structure, which returns data about the mouse's position when it reaches any of the window's margins, there are also the following structures: MLESEARCHDATA, MLECTLDATA, MLEFORMATRECT, and MLEOVERFLOW.

**Table 7.26 The MLN\_ Notification Codes of the Class WC\_MLE**

<i>Notification Code</i>	<i>Value</i>	<i>Description</i>
MLN_OVERFLOW	0x0001	Returned to the owner window when the user tries to insert more character than those that can fit in the <i>mle</i> window.
MLN_PIXHORZOVERFLOW	0x0002	The maximum number of characters that can fit in a previously formatted <i>mle</i> window has been exceeded.
MLN_PIXVERTOVERFLOW	0x0003	The maximum number of characters that can fit in a previously formatted <i>mle</i> window has been exceeded.
MLN_TEXTOVERFLOW	0x0004	Operations pertaining to text handling have exceeded some preset limit.
MLN_VSCROLL	0x0005	Vertical <i>scrolling</i> took place; also indicates the number of the first visible line.
MLN_HSCROLL	0x0006	Informs the owner window that horizontal <i>scrolling</i> occurred; also indicates how many pixels have become invisible on the X axis.
MLN_CHANGE	0x0007	The contents of the <i>mle</i> window have changed due to insertion or deletion of text.
MLN_SETFOCUS	0x0008	The <i>mle</i> has acquired <i>focus</i> .
MLN_KILLFOCUS	0x0009	The <i>mle</i> has lost <i>focus</i> .
MLN_MARGIN	0x000a	Informs the owner of the <i>mle</i> that the mouse has been moved over one of the margins of the window: The data returned in the structure MLEMARGSTRUCT establishes on which of the four sides the cursor is, the message associated with the mouse movement, and the character that is closest to the mouse's position.
MLN_SEARCHPAUSE	0x000b	Issued periodically by the <i>mle</i> when it is engaged in the execution of the MLM_SEARCH message; this allows the application to interrupt the search operation.
MLN_MEMERROR	0x000c	Issued when a direct action upon the <i>mle</i> causes the text limit or the formatting rectangle to be exceeded.
MLN_UNDOOVERFLOW	0x000d	Issued to notify that it was impossible to perform a text editing action (undo) because it would cause the text limit to be exceeded.
MLN_CLPBDFAIL	0x000f	Issued to inform that the Clipboard was not able to accommodate the text that was sent to it by the <i>mle</i> .

## The MLECTLDATA Structure

The MLECTLDATA structure contains data about the formatting of an *mle*, the position of the cursor, and the definition of the selected text.

```
typedef struct _MLECTLDATA
{ // mlectl
  USHORT cbCtlData ;
  USHORT afIEFormat ;
  ULONG cchText ;
  IPT iptAnchor ;
  IPT iptCursor ;
  LONG cxFormat ;
  LONG cyFormat ;
  ULONG afFormatFlags ;
} MLECTLDATA ;

typedef MLECTLDATA *PMLECTLDATA ;
```

Table 7.27 describes each member in the MLECTLDATA structure.

When an *mle* is created, the programmer can define some of its behavioral and structural traits without having to resort to sending specific messages at later times.

**Table 7.27 Members of the MLECTLDATA Structure**

<i>Member</i>	<i>Description</i>
cbCtlData	Size of the structure.
afIEFormat	Format of execution of <i>import/export</i> operations. The possible options are: <ul style="list-style-type: none"> <li>MLFIE_CFTEXT 0</li> <li>MLFIE_NOTRANS 1</li> <li>MLFIE_WINFMT 2</li> <li>MLFIE_RTF 3</li> </ul>
cchText	Maximum text size, expressed in bytes.
iptAnchor	Defines the position, expressed in bytes, of the first character of the selected text.
iptCursor	Indicates the cursor's position.
cxFormat	Width, expressed in pixels, of the format rectangle.
cyFormat	Height, expressed in pixels, of the format rectangle.
afFormatFlags	Defines how the format rectangle should be handled. The possible options are: <ul style="list-style-type: none"> <li>MLFFMTRECT_LIMITHORZ</li> <li>MLFFMTRECT_LIMITVERT</li> <li>MLFFMTRECT_MATCHWINDOW</li> <li>MLFFMTRECT_FORMATRECT</li> </ul>



Listing 7.9 is an example of how to use the services of the class WC\_MLE to create a simple, but functional, word processor for PM. The governing logic of the application allows you to position an *mle* window directly over the client of the application, and to handle all possible cut and paste operations through appropriate messages, and vary accordingly the status of the menu items in the Edit menu.

## The Class WC\_NOTEBOOK

Let's now get into the novelties of OS/2 2.1 and examine the last five predefined classes. As you will have noticed, the classes have been of growing complexity and flexibility in terms of the actions they support. WC\_NOTEBOOK is similar to a three-dimensional, spiral-bound notebook with tabs. The distinctive characteristic of a notebook is the notebook *pages*. As you will discover soon, the designer can use and interact only marginally with the WC\_NOTEBOOK class. The only possible operations with a notebook are establishing the number of pages needed, and associating one or more windows with each page. The examination of the notebook is helped by scroll arrows at the bottom of every page, or by acting directly on the tabs along the side (Figure 7.23).

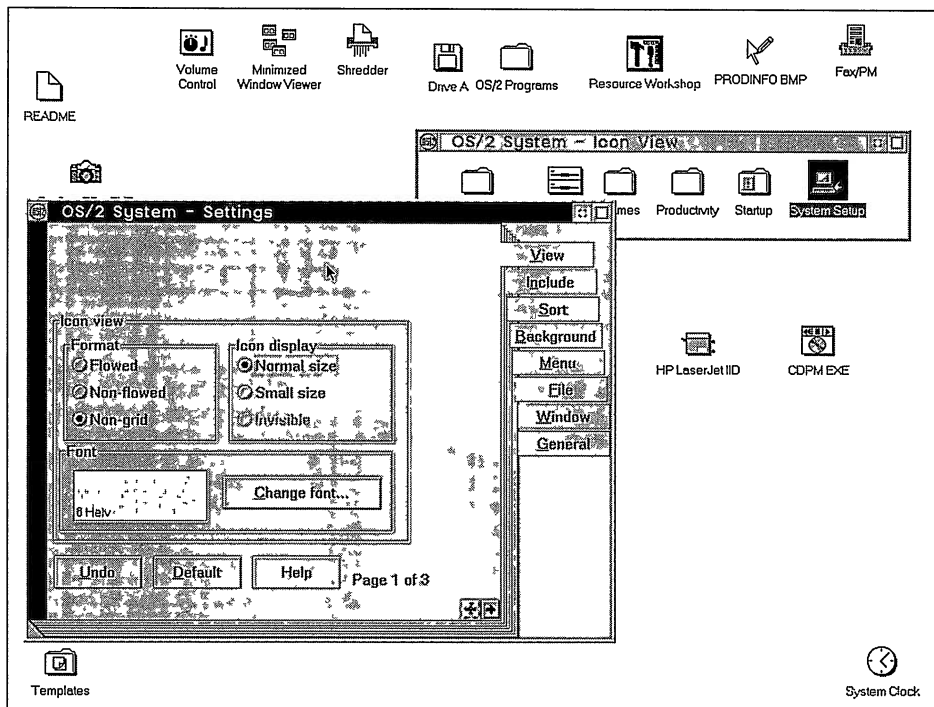


Figure 7.23 Object settings in WPS are always controlled through a window of the class WC\_NOTEBOOK.

The use of notebooks in WPS has some advantages. All users, even newcomers, are familiar with this kind of control, and *expect* to find it in applications. Before proceeding any further, try using the SNOOPER utility to explore the *Settings* windows of a WPS object. You will be surprised to discover that the WC\_NOTEBOOK class window takes up only a limited space compared to the overall dimensions of the object. The visible page is always a window belonging to one of the predefined classes (almost invariably WC\_FRAME) or some class registered in the application. Let's see, step by step, how such a window is created with *WinCreateStdWindow()*:

```
...
hwndFrame = WinCreateStdWindow(  HWND_DESKTOP,
                                WS_VISIBLE,
                                &u1FCFflags,
                                WC_NOTEBOOK,
                                szWindowTitle,
                                BKS_XXX,
                                NULLHANDLE,
                                RS_ALL,
                                &hwndNotebook) ;
...
```

Are you still there? Don't be afraid, there's nothing magic going on here. The effect produced by the above code fragment is generating a frame window, with all controls implied by the FCF\_ combinations (not described here), and with a window belonging to the class WC\_NOTEBOOK. A concrete result of generating a window in this way is that you don't have to register a window class and that you don't need a window procedure. In practice, the application is limited to the sole *main()* function. We won't list the sample code here, but you can try to create it, as this is an excellent exercise in using the predefined classes.

The presence of a menu bar will certainly cause you some problems! The WM\_COMMAND messages will reach the window procedure of the WC\_NOTEBOOK class without passing through any portion of the source code. There are two possible solutions:

- Interpose between the frame window and the notebook a client belonging to some class registered in the code
- Subclassing the frame

The first way is the one most commonly adopted, although it does have the disadvantage of increasing the amount of code and inducing too many windows. Therefore, although we have not yet seen what *subclassing* really is, we will use this second method. The code presented in Listing 7.10 and in subsequent listings performs a subclassing of the frame immediately after the creation of the main window.

```
...
pfnewp = WinSubclassWindow( hwndFrame, FrameWndProc) ;
WinSendMsg( hwndFrame, WM_PASSPROC, MPFROMP( pfnewp), 0L) ;
...
```

In the listing you can now see the *FrameWndProc()*, the window procedure of the application's frame window. This takes care of intercepting the notification codes issued by the notebook to its owner, and also the *WM\_COMMAND* possibly generated by the menu bar. If you have read the previous sentence carefully, and remember the considerations drawn in Chapter 4 about ownership, you should now be able to see a problem. What? The client produced by *WinCreateStdWindow()* does not use the frame as its owner. It is therefore necessary to enforce this condition. The preceding code fragment must thus be improved with a call to *WinSetOwner()*:

```
...
WinSetOwner( hwndNotebook, hwndFrame ) ;
...
```

At this point all the pieces are in place. The window of class *WC\_NOTEBOOK* has as its ID the value of *FID\_CLIENT*, and it is handled by the frame as far as basic operations are concerned (resizing, painting, and so on). However, subclassing is not limited to the frame alone; it can also pertain to the notebook itself by following exactly the same scheme, and changing only the involved handles.

Now, let's get back to the detailed syntax needed for creating a notebook. In the style parameter of the client you can include the defines introduced by the *BKS\_* prefix: These are all the styles that are specific to the class *WC\_NOTEBOOK* (Table 7.28).

**Table 7.28 The BKS\_ Styles of the Class WC\_NOTEBOOK**

<i>3D Effect Styles</i>	<i>Value</i>	<i>Description</i>
<i>BKS_BACKPAGESBR</i>	0x00000001	Lower pages are visible to the right
<i>BKS_BACKPAGESBL</i>	0x00000002	Lower pages are visible to the left
<i>BKS_BACKPAGESTR</i>	0x00000004	Higher pages are visible to the right
<i>BKS_BACKPAGESTL</i>	0x00000008	Lower pages are visible to the left
<i>Tab Position Styles</i>	<i>Value</i>	<i>Description</i>
<i>BKS_MAJORTABRIGHT</i>	0x00000010	To the right
<i>BKS_MAJORTABLEFT</i>	0x00000020	To the left
<i>BKS_MAJORTABTOP</i>	0x00000040	Top border
<i>BKS_MAJORTABBOTTOM</i>	0x00000080	Bottom border
<i>Tab Type Styles</i>	<i>Value</i>	<i>Description</i>
<i>BKS_SQUARETABS</i>	0x00000000	Tab with sharp angles
<i>BKS_ROUNDEDTABS</i>	0x00000100	Tab with rounded angles
<i>BKS_POLYGONTABS</i>	0x00000200	Tab with polygonal angles

(continued)



Table 7.28 (Continued)

<i>Binding Type Styles</i>	<i>Value</i>	<i>Description</i>
BKS_SOLIDBIND	0x00000000	Solid
BKS_SPIRALBIND	0x00000400	Spiral
<i>Information Line Styles</i>	<i>Value</i>	<i>Description</i>
BKS_STATUSTEXTLEFT	0x00000000	Left aligned text
BKS_STATUSTEXTRIGHT	0x00001000	Right aligned text
BKS_STATUSTEXTCENTER	0x00002000	Centered text
<i>Tab Text Styles</i>	<i>Value</i>	<i>Description</i>
BKS_TABTEXTLEFT	0x00000000	Left aligned text
BKS_TABTEXTRIGHT	0x00004000	Right aligned text
BKS_TABTEXTCENTER	0x00008000	Centered text

By means of these styles, the programmer can define the perspective of the notebook, the position of tabs, their shape, the kind of text alignment, the binding mode, and the text alignment in the information line. This last item is the area used to display a text string that summarizes for the user the purpose of the operations supported by the notebook. The *Settings* objects of WPS do not use an information line; that explains why it is not in Figure 7.23.

## Filling a Notebook

Styles will provide only part of the aesthetic, functional, and structural characteristics of a notebook, at least compared to what you are used to in WPS objects. What remains yet undefined is the insertion of text into the page tabs, the definition of the number of pages, and most important, filling in the pages with information. Let's proceed with order. When a WC\_NOTEBOOK class window is created, the programmer can define the position of the page tabs and their shape. Their size, contents, and tasks performed will be specified by sending the messages BKM\_SETDIMENSIONS and BKM\_SETTABTEXT (Table 7.29).

A notebook has two kinds of page tabs: primary and secondary, pertaining to primary and secondary arguments, respectively. The message BKM\_SETDIMENSIONS is used to set the size of both kinds of page tabs, and of the paging arrows.

**Table 7.29 The Messages of the Class WC\_NOTEBOOK**

<i>Message</i>	<i>Value</i>	<i>Description</i>
BKM_CALCPAGERECT	0x0353	Computes the size of a notebook page.
BKM_DELETEPAGE	0x0354	Removes one or more pages from the notebook.
BKM_INSERTPAGE	0x0355	Inserts a page into the notebook.
BKM_INVALIDATETABS	0x0356	Enforces the repainting of the contents of a notebook.
BKM_QUERYPAGECOUNT	0x0358	Returns the number of pages in the notebook.
BKM_TURNTOPAGE	0x0357	Brings to foreground the page indicated by <i>pageID</i> .
BKM_QUERYPAGEID	0x0359	Returns the <i>pageID</i> .
BKM_QUERYPAGEDATA	0x035a	Returns information inserted in the 4 reserved bytes of each page.
BKM_QUERYPAGEWINDOWHWND	0x035b	Returns the handle of the window associated with a page.
BKM_QUERYTABBITMAP	0x035c	Returns the bitmap of a page tab.
BKM_QUERYTABTEXT	0x035d	Returns the text of a page tab.
BKM_SETDIMENSIONS	0x035e	Sets the size of page tabs and paging arrows.
BKM_SETPAGEDATA	0x035f	Inserts information defined by the application in the 4 reserved bytes available to each page.
BKM_SETPAGEWINDOWHWND	0x0360	Associates a window to a page in the notebook.
BKM_SETSTATUSLINETEXT	0x0361	Inserts text in the information line.
BKM_SETTABBITMAP	0x0362	Inserts a bitmap in a page tab.
BKM_SETTABTEXT	0x0363	Inserts text in a page tab.
BKM_SETNOTEBOOKCOLORS	0x0364	Sets the colors of the various pieces of a notebook.
BKM_QUERYPAGESTYLE	0x0365	Returns the attributes of the page indicated with the message BKM_INSERTPAGE.
BKM_QUERYSTATUSLINETEXT	0x0366	Returns the text written in the information line.

BKM_SETDIMENSIONS	0x035e	<i>Description</i>
mp1	USHORT usWidth USHORT usHeight	Width of the object Height of the object
mp2	USHORT usType	One of the following values: BKA_MAJORTAB BKA_MINORTAB BKA_PAGEBUTTON
Return Value	BOOL fSuccess	Result of the operation

The sizes that need to be packed into mp1 are always expressed in pixels. The sizing of a page tab can take place immediately after the creation of a window belonging to the WC\_NOTEBOOK class; the addition of text requires that you first define the page handling strategy. In fact, at least as far as the primary page tabs are concerned, there is a strong correlation with the page strategy.

## Inserting a Page

Despite the three-dimensional look of a notebook, its true “depth” comes from the number of pages it actually contains, not by the number of pages drawn on the screen. Inserting a page requires, in the first place, that you identify the correct position and assign it an appropriate ID. The message BKM\_INSERTPAGE performs all the following operations:

BKM_INSERTPAGE	0x0355	<i>Description</i>
mp1	ULONG ulPageID	ID of the page to be inserted
mp2	USHORT usPageStyle	One or more of the following values: BKA_AUTOPAGESIZE BKA_STATUSTEXTON BKA_MAJOR BKA_MINOR
	USHORT usPageOrder	One of the following values: BKA_FIRST BKA_LAST BKA_NEXT BKA_PREV
Return Value	ULONG ulPageID	ID of the inserted page or NULL in case of failure

The pages in a notebook are identified by an ID, just like the items in a listbox. This value is either assigned by the programmer through mp1 or it is returned in response to a BKM\_INSERTPAGE message. This second solution is more convenient, since it delegates this task to the notebook itself. In general, mp1 is set to NULL. In the first USHORT of mp2 you have to specify one or more attributes for defining the characteristics of the pages. It is at this point that the programmer can decide on a primary

page tab (BKA\_MAJOR) or a secondary page tab (BKA\_MINOR) and the information line (BKA\_STATUSTEXTON). The flag BKA\_AUTOPAGESIZE is almost always set because it makes it a lot easier to handle the page's output as it ensures that the notebook itself will redraw the page as a result of resizing the window. If it is not set, the application instead receives the notification code BKN\_NEWPAGESIZE and tells the owner that it will take care of redrawing the page.

In the second USHORT of mp2 there is a flag used to indicate the position of the new page. The two defines BKA\_FIRST and BKA\_LAST are most often used, because they will always insert a new page as the first or last one in the stack of pages. Resorting to BKA\_PREV and BKA\_NEXT is possible if you have indicated a *pageID* in mp1. In general, especially during initialization of a notebook, a common strategy is inserting a page at a time, starting with the first one, and then using BKA\_LAST. For any following dynamic insertions, once the *pageIDs* are already known, you may have to force an insertion at a specific point on the Z axis.

Note that the action performed by BKM\_INSERTPAGE is limited to inserting a page, then returning to the application with an essential item, that is, the *pageID*. Once you have the *pageID*, it is possible to define the text in the page tabs (primary or secondary) previously defined (BKM\_SETTABTEXT), or insert some text in the information line if present (BKM\_SETSTATUSLINETEXT), and finally associate the notebook page with a window (BKM\_SETPAGEWINDOWHND).

	BKM_SETTABTEXT	0x0363	<i>Description</i>
mp1	ULONG	ulPageID	ID of the page fitted with a page tab
mp2	PSZ	pszString	Text to insert into the page tab
Return Value	BOOL	fSuccess	Success or failure of the operation

As an alternative to text strings, a page tab can become even more intuitive if you resort to an image and insert it with BKM\_SETTABBITMAP as shown in listing 7.10. Now, there is not much left to do, other than filling the page with windows that will allow the user to perform the designated operations. Almost always, these windows will be a collection of controls, as you might have inferred by looking into any *Settings* notebook of WPS objects. As you will see in the next chapter, an ideal solution is implementing a dialog window and associating it with a notebook page. Basically, a dialog is a container of controls. It is sufficient to specify the handle of the dialog in mp2 in the message BKM\_SETPAGEWINDOWHND in order to get the results you want, while in mp1 you must always indicate the *pageID*.

	BKM_SETPAGEWINDOWHND	0x0360	<i>Description</i>
mp1	ULONG	ulPageID	ID of a page equipped with a page tab
mp2	HWND	hwnd	Handle of the window to be associated with the page
Return Value	BOOL	fSuccess	Success or failure of the operation

**Table 7.30 The Notification Codes of the Class WC\_NOTEBOOK**

<i>Notification Code</i>	<i>Value</i>	<i>Description</i>
BKN_PAGESELECTED	130	A page has been selected.
BKN_NEWPAGESIZE	131	The page has been resized.
BKN_HELP	132	Help has been requested in a notebook.
BKN_PAGEDLETED	133	A page has been deleted.

Any kind of window will work correctly in a notebook page, including windows belonging to predefined classes, and windows registered by the application. In some cases, in place of a dialog, you will have the handle of a window that has been registered by the code together with several child windows. In fact, albeit the *Settings* of WPS establish a valid reference model for notebooks, you are not obliged to stick to using this class of windows in the system shell. For instance, the combination of a notebook bound along the upper edge with a *valueset* is an excellent couple for designing calendars or planners.

## *Associating Information with a Page*

The similarities between pages in notebooks and items in listboxes also extend to the 4 extra bytes per page available to the designer. To access this area you need to issue the messages BKM\_SETPAGEDATA and BKM\_QUERYAGEDATA. Almost invariably, the contents will be the handle of some window to which the notebook page refers.

## *Notification Codes*

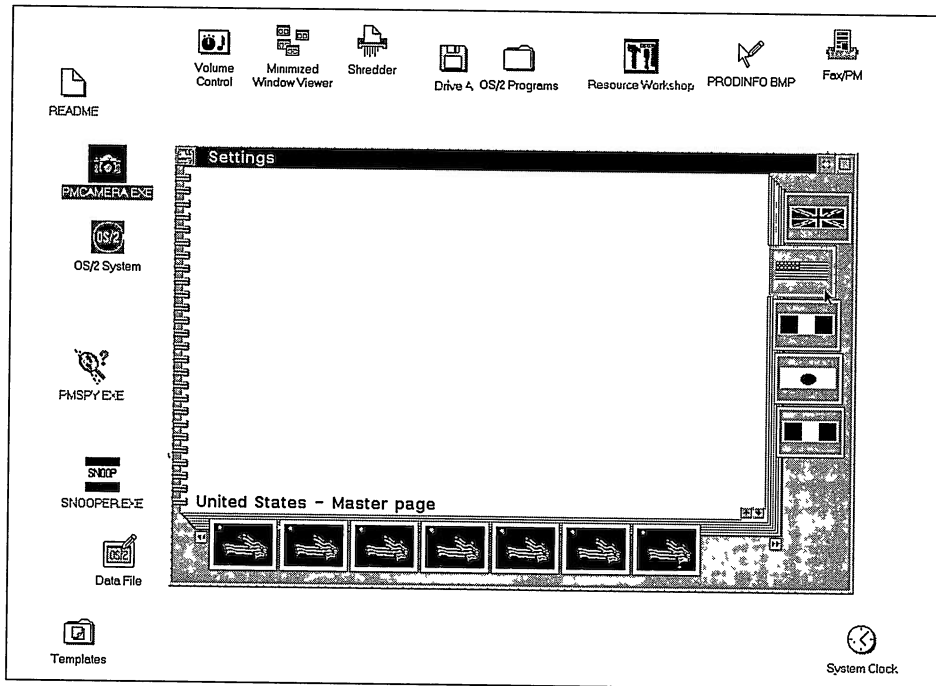
The syntax of the WC\_NOTEBOOK is completed with the notification codes listed in Table 7.30.

## *A First Try*



In Listing 7.10 you can see the source code of the NOTEBOOK application, a first try at using notebooks. The window is well equipped (Figure 7.24), although it has no contents in any pages (in Chapter 13 you will find the full example).

In NOTEBOOK the code performs *subclassing*, not of the frame, but of the notebook *itself*, because at the current stage there is a small bug in the API. In fact, closing the window by a double-click on the titlebar menu icon, will not cause the complete destruction of the frame, as it will leave all the application windows on the screen. Therefore it is necessary to intercept the WM\_CLOSE message and handle it appropriately to bypass this problem.



**Figure 7.24** The NOTEBOOK application is equipped with all structural elements available to this kind of window, but it is missing any kind of window associated with its pages.

## The Class WC\_CONTAINER

One of the most interesting aspects of the interface of OS/2 2.1 is folders. These top-level windows belong to the class WC\_FRAME and have as client area a representative of the class WC\_CONTAINER (this information was gathered through SNOOPER). Hereafter, the terms container and folders are used interchangeably and both identify a window of the class WC\_CONTAINER. Knowing what WPS folders can do, it is evident that this class must be interesting and powerful. Surprisingly, this complexity does not cause any particular difficulty in creating and using a container. The only inconvenience is in the sheer volume of data that needs to be handled before a WC\_CONTAINER class window becomes operative. Examining the API of this class you will see that it is a new breed of window; only a few styles, a limited number of messages, great significance of notification codes, and, above all, lots of data structures.

### *The Styles of the Class WC\_CONTAINER*

The first three styles of the class WC\_CONTAINER (Table 7.31) define the selection rules within a window of this class. The best solution is to set the CCS\_EXTENDSEL style, in

Table 7.31 The Styles of the Class WC\_CONTAINER

<i>Style</i>	<i>Value</i>	<i>Description</i>
CCS_EXTENDSEL	0x00000001L	Allows you to select with the mouse more than one object at a time.
CCS_MULTIPLESEL	0x00000002L	Allows you to select more than one object at a time.
CCS_SINGLESEL	0x00000004L	Allows you to select only one object at a time.
CCS_AUTOPOSITION	0x00000008L	Contained objects are positioned automatically in icon view.
CCS_VERIFYPOINTERS	0x00000010L	Verifies that the pointers used in handling a container actually belong to the linked list maintained by the system.
CCS_READONLY	0x00000020L	Prevents the user from changing any piece of text in the container.
CCS_MINIRECORDCORE	0x00000040L	Data regarding the container record are always of the type MINIRECORDCORE.

order to enable selection of multiple objects at the same time, simply by dragging the mouse over them (if they are adjacent), or interacting with the keyboard (if they are distributed in diverse positions). Almost always it will be convenient to set the CCS\_AUTOPOSITION style, so as to delegate to the container the placement of its contents.

The style CCS\_READONLY will prevent the user from changing the text strings, but it will not affect any other functionality of the window. The last two styles, CCS\_VERIFYPOINTERS and CCS\_MINIRECORDCORE, concern mostly the internal handling of the objects present in the container.

## *The Logic of the WC\_CONTAINER Class*

If you take a look at Figure 7.25 you can see that *OS/2 System* appears in three distinct modes: *icon view*, *tree view*, and *detail view*. These are three of the five operating modes of a WC\_CONTAINER class window.

WPS will support only these three out of five display modes of the contents of a container. Table 7.32 presents the complete list.

WPS supports all five display modes listed in Table 7.32. The two that do not appear in the second level menu introduced by Open (*Name view* and *Text view*) are easy to implement by eliminating the icons or selecting the radiobutton *Flowed* in each notebook Settings.

As its name suggests, the purpose of a container is that of containing objects. These can be text strings, icons with text labels, and/or bitmaps. Due to its complexity, all

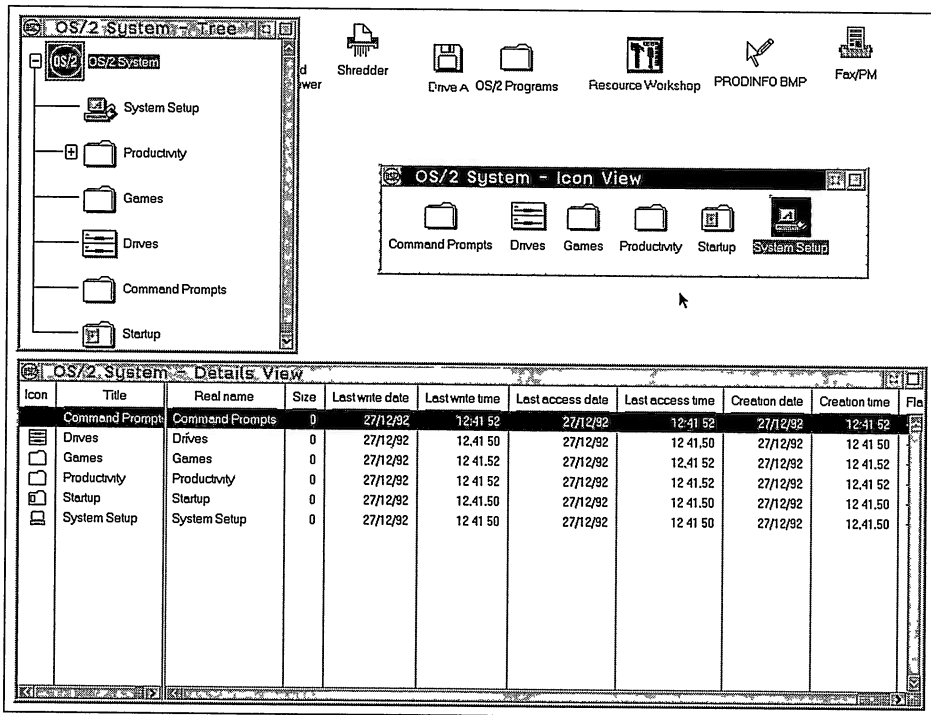


Figure 7.25 The three display modes of a container, according to WPS.

information cannot be inserted directly in a container at creation time, as was the case, for instance, of text strings in a listbox. The sheer volume of data is much greater, and the data itself is much more complex. Therefore, an important consideration for this class is that you will need to prepare data in the application before passing it to a container. Furthermore, due to the high degree of versatility of the WC\_CONTAINER, most features are not expressed under the form of styles, as we have seen in Table 7.32, but almost invariably use specific data structures and messages (Table 7.33).

Table 7.32 The Five Output Modes of a Container

<i>Mode</i>	<i>Description</i>
Icon view	Displays icons and/or bitmaps with a text label underneath.
Name view	Displays icons and/or bitmaps with a text label to their right.
Text view	Displays text strings only.
Tree view	Presents a tree view of the contents of a container. Three variations are allowed: tree text, tree name, and tree icon.
Details view	For each item contained, several pieces of information will be displayed in different columns headed by a text label or an image.



**Table 7.33 The Messages of the Class WC\_CONTAINER**

<i>Message</i>	<i>Value</i>	<i>Description</i>
CM_ALLOCDETAILFIELDINFO	0x0330	Allocates memory for one or more FIELDINFO structures.
CM_ALLOCRECORD	0x0331	Allocates memory for one or more RECORDCORE structures.
CM_ARRANGE	0x0332	Rearranges the objects in a container when in icon view.
CM_ERASERECORD	0x0333	Erases a record due to a <i>move</i> operation.
CM_FILTER	0x0334	Filters the contents of a container so as to display only a specific portion of it.
CM_FREEDetailFIELDINFO	0x0335	Frees the memory associated with one or more FIELDINFO structures.
CM_FREERECORD	0x0336	Frees the memory associated with one or more RECORDCORE or MINIRECORDCORE structures.
CM_HORZSCROLLSPLITWINDOW	0x0337	Performs a horizontal scrolling of a container in the Detail mode and furnished with a <i>split window</i> .
CM_INSERTDETAILFIELDINFO	0x0338	Inserts one or more FIELDINFO structures in the container.
CM_INSERTRECORD	0x0339	Inserts one or more RECORDCORE structures in the container.
CM_INVALIDATEDetailFIELDINFO	0x033a	Forces the container to perform a refresh of its contents.
CM_INVALIDATERECORD	0x033b	Forces the container to perform a refresh of its contents.
CM_PAINTBACKGROUND	0x033c	Colors the container's background.
CM_QUERYCNRINFO	0x033d	Returns a pointer to a CNRINFO structure.
CM_QUERYDETAILFIELDINFO	0x033e	Returns a pointer to the required FIELDINFO structure.
CM_QUERYDRAGIMAGE	0x033f	Returns a handle to the icon or to the bitmap of the object being dragged.
CM_QUERYRECORD	0x0340	Returns a pointer to the required RECORDCORE structure.
CM_QUERYRECORDEMPHASIS	0x0341	Returns the record that presents a specific emphasis attribute.
CM_QUERYRECORDFROMRECT	0x0342	Returns the record that is enclosed by the indicated rectangle.
CM_QUERYRECORDRECT	0x0343	Returns the rectangle of the indicated record.
CM_QUERYVIEWPORTRECT	0x0344	Returns the rectangle containing the container's output area.

*(continued)*

Table 7.33 (Continued)

<i>Message</i>	<i>Value</i>	<i>Description</i>
CM_REMOVEDTAILFIELDINFO	0x0345	Removes one or more FIELDINFO structures from the container.
CM_REMOVERECORD	0x0346	Removes one or more RECORDCORE structures from the container.
CM_SCROLLWINDOW	0x0347	Performs a scrolling of the whole container window.
CM_SEARCHSTRING	0x0348	Returns a pointer to the RECORDCORE structure containing the indicated search string.
CM_SETCNRINFO	0x0349	Changes or sets the container information.
CM_SETRECORDEMPHASIS	0x034a	Sets the emphasis attributes of a record.
CM_SORTRECORD	0x034b	Sorts the contents of a container.
CM_OPENEDIT	0x034c	Opens the <i>mle</i> used for editing the title of an object in the container.
CM_CLOSEEDIT	0x034d	Closes the <i>mle</i> used for editing the title of an object in the container.
CM_COLLAPSETREE	0x034e	Collapses the tree structure.
CM_EXPANDTREE	0x034f	Expands the tree structure.
CM_QUERYRECORDINFO	0x0350	Used to update the records of objects contained in several containers of the same process.

## Creating a Container

The expression *creating a container* does not only mean calling *WinCreateWindow()*; it also includes all the initialization operations that are necessary. You can proceed in the following way for a container in *icon view*:

- Allocate the memory area necessary for describing all items to be inserted into the container by sending the message CM\_ALLOCRECORD.
- Fill in a number of RECORDCORE structures equal to the number of items to be inserted into the container.
- Fill in a RECORDINSERT structure.
- Pass all information pertaining to the RECORDCORE structures to the RECORDINSERT structures by sending the message CM\_INSERTRECORD.
- Customize the look of the container by acting appropriately on the members of the CNRINFO structure, and then send the message CM\_SETCNRINFO.

The list of operations just described is not valid in absolute terms, and should not be interpreted strictly. For instance, if the container was created with the `CCS_MINIRECORDCORE` flag, you might want to use a `MINIRECORDCORE` structure in place of a `RECORDCORE` structure. The best technique is to fill in all pieces of data in a `RECORDCORE` structure, then those of the `RECORDINSERT` structure, and lastly, sending the `CM_INSERTRECORD` message. However, nothing prevents you from combining a `RECORDCORE` and a `RECORDINSERT` and then using the `CM_INSERTRECORD`. If the container operates in *details view*, then the structures that have to be used are `FIELDINFO` and `FIELDINFOINSERT`. Naturally, we will not describe all possible variations here. What is important to understand is the basic sequence of operations that need to be performed. The descriptions in the next few paragraphs refer to a container in *icon view* mode.

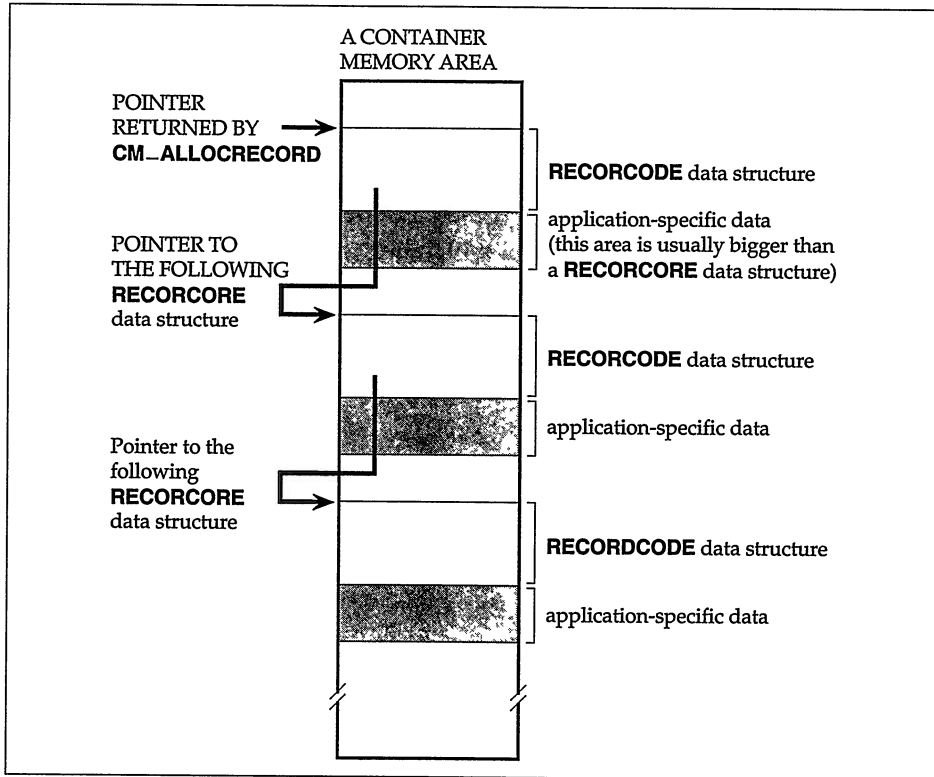
## The Objects of a Container

Apart from their look (icon, bitmap, text, or any combination), each item is inserted in a container like a record. As these are composite data structures, each record corresponds to a memory area handled by the container itself, although initially it must be set up by the application. Sending the message `CM_ALLOCORECORD` will allocate a memory area large enough to describe all objects (records) that need to be catered for initially.

		<i>Description</i>
<code>CM_ALLOCORECORD</code>	<code>0x0331</code>	
<code>mp1</code>	<code>ULONG cbRecordData</code>	Extra bytes to be allocated for each single record
<code>mp2</code>	<code>USHORT nRecord</code>	Number of <code>RECORDCORE</code> (or <code>MINIRECORDCORE</code> ) structures
Return Value	<code>PRECORDCORE prec</code>	Address of an array of <code>RECORDCORE</code> (or <code>MINIRECORDCORE</code> ) structures

In `mp2` you must indicate the number of items that you want to insert into the container. This is only the initial quantity, which you can later change while you are using the window. The `RECORDCORE` structure is 56 bytes large. The class `WC_CONTAINER` will allocate a memory area large enough to contain the overall sum of the required `RECORDCORE` structures, and will then sub-allocate fixed-sized blocks. In addition to these 56 bytes, memory allocation will also account for the bytes requested by the programmer in `mp1`. Figure 7.26 depicts the memory area allocated by the `WC_CONTAINER` class in response to the message `CM_ALLOCORECORD`.

The value in `mp1` corresponds to the additional memory area for each single item. Its position comes in memory immediately after the last member of the `RECORDCORE` structure. The `RECORDCORE` blocks, though, are not adjacent to one another, with or without the additional bytes. In fact, a `RECORDCORE` structure contains a `PRECORDCORE` type member pointing to the address of the next `RECORDCORE` structure in the list. So, what we actually have here is a single-linked list originally handled by the `WC_CONTAINER` class.



**Figure 7.26** Scheme of memory allocation for handling the RECORDCORE structures.

It is vital to allocate extra space for each item. As soon as you become familiar with this class, you will appreciate this advice. Managing the contents of a container is distributed partially to the window itself and the application. This is why it is worthwhile to define a customized data structure that contains as its first member a RECORDCORE, followed by application specific values. In Listing 7.11 we define a structure called APPREC; any other name would do as well. The only caution to be aware of is that of placing the RECORDCORE structure as the first member of the customized structure:

```
typedef struct APPREC
{
    RECORDCORE rec ;
    PRECORDCORE precParent ;
    LONG lType ;
    HWND hpopup ;
} APPREC ;

typedef APPREC * PAPPREC ;
```

In this specific case, the additional information consists of the address of a RECORDCORE structure, a LONG encoding the type of object contained, and a window handle. The WPS interface has accustomed us to consider the contents of containers as objects, applications, data files, or even physical devices. In many cases an icon represents some element of the file system. However, nothing prevents a container from holding abstract objects, representing application specific information. So, you might think of icons representing various methods for performing statistical analysis, file conversions, or just about anything else. Notwithstanding their specific nature they are RECORDCORE data from the WC\_CONTAINER perspective. Furthermore, an application must be able to manage and support all operations performed by the user. Associating standard record information with application-specific data tied directly to the object is the best way of writing this kind of code.

Often it is useful to declare a pointer to the customized structure (PAPPREC), and another to the standard structure (PRECORDCORE): this will usually make the code simpler to write and understand:

```
...
PRECORDCORE prec ;
PAPPREC papprec ;
...
prec = WinSendMessage( hwndCnr, CM_ALLOCRECORD,
                      MPFROMLONG( sizeof( APPREC ) - sizeof( RECORDCORE ) ),
                      MPFROMSHORT( NUMOBJECTS ) ) ;
papprec = (PAPPREC) prec ;
...
```

Once you have the starting address of the block, you only need to fill in the appropriate members of the RECORDCORE structure, as described in Table 7.34:

```
typedef struct _RECORDCORE
{ // recc
  ULONG cb ;
  ULONG flRecordAttr ;
  POINTL ptlIcon ;
  struct _RECORDCORE *preccNextRecord ;
  PSZ pszIcon ;
  HPOINTER hptrIcon ;
  HPOINTER hptrMiniIcon ;
  HBITMAP hbmBitmap ;
  HBITMAP hbmMiniBitmap ;
  PTREEITEMDESC pTreeItemDesc ;
  PSZ pszText ;
  PSZ pszName ;
  PSZ pszTree ;
} RECORDCORE ;

typedef RECORDCORE *PRECORDCORE ;
```

Table 7.34 Data Members of the Structure RECORDCORE

<i>Member</i>	<i>Description</i>
cb	Size of the structure.
flRecordAttr	Container attributes; the possible options are: CRA_SELECTED           0x00000001L CRA_TARGET             0x00000002L CRA_CURSORED          0x00000004L CRA_INUSE              0x00000008L CRA_FILTERED           0x00000010L CRA_DROPONABLE        0x00000020L CRA_RECORDREADONLY   0x00000040L CRA_EXPANDED           0x00000080L CRA_COLLAPSED         0x00000100L CRA_SOURCE             0x00004000L
ptlIcon	Position of the icon representing a record in a container in <i>icon view</i> mode.
preccNextRecord	Address of the next RECORDCORE structure.
pszIcon	Text in the <i>icon view</i> mode.
hptrIcon	Handle of the icon to be displayed in the container.
hptrMiniIcon	Handle of a reduced-sized icon to be displayed in <i>details view</i> mode or with shrunk icons.
hbmBitmap	Handle of the bitmap to be displayed for representing an object (an alternative to the handle of the icon).
hbmMiniBitmap	Handle of a reduced-sized bitmap to be displayed in <i>details view</i> mode or with shrunk objects.
pTreeItemDesc	Address of a TREEITEMDESC structure containing the bitmaps to be displayed to represent the tree structure in <i>tree view</i> mode.
pszText	Text for the <i>text view</i> (CV_TEXT) mode.
pszName	Text for the <i>name view</i> (CV_NAME) mode.
pszTree	Text for the <i>tree view</i> (CV_TREE) mode.

Of all members in the RECORDCORE structure, two are truly important for the *icon view* mode: the handle to the icon (hptrIcon) and the text string (pszIcon). For all other members it is possible not to specify any value, because the memory area is initialized to zero due to an internal call to *DosAllocMem()*.

You must be particularly careful with text strings in the RECORDCORE structure. The class WC\_CONTAINER does not provide any way to store strings permanently and safely. This task is delegated to the application. In fact, all members of a RECORDCORE structure that in some way deal with a string simply require the address of some memory location. The programmer is responsible for allocating a memory area that is accessible

to the class and that always contains text strings appropriate for each single item. In the case of the FOLDER example, this problem has been solved simply by resorting to the resource file and to string constantly stored at the same position. A better approach would be to take advantage of the extra space allocated through CM\_ALLOCRECORD, and store there all strings of each object. In Chapter 13 you will find an example that implements such a solution.

To store application-specific information, you use the PAPPREC pointer. The filling of the RECORDCORE structure is repeated for each item to be inserted by updating the pointer in this way:

```
prec = prec -> precNextRecord ;
```

Before the actual insertion of the object in the container, you must prepare a RECORDINSERT structure (described in Table 7.35).

```
typedef struct _RECORDINSERT
{ // recins
  ULONG cb ;
  RECORDCORE pRecordOrder ;
  RECORDCORE pRecordParent ;
  ULONG fInvalidateRecord ;
  ULONG zOrder ;
  ULONG cRecordsInsert ;
} RECORDINSERT ;

typedef RECORDINSERT *PRECORDINSERT ;
```

**Table 7.35 Members of the RECORDINSERT Structure**

<i>Member</i>	<i>Description</i>
cb	Size of the structure.
pRecordOrder	Defines the insertion order of new records, with respect to those already existing in the container. The defines CMA_FIRST and CMA_END will position them, respectively, at the start or at the end of the existing list. The address of a RECORDCORE structure is used as the precise indication of where the insertion is to take place in the list.
pRecordParent	Is valid only if pRecordOrder is set to CMA_FIRST or CMA_END, and then indicates the parent record of those being inserted.
fInvalidateRecord	Indicates if the new records are to be displayed immediately (TRUE) or later when there will be a specific request for repainting.
zOrder	Indicates the position of new records according to the Z-order. CMA_TOP and CMA_BOTTOM are the only two options available.
cRecordInsert	Number of RECORDCORE (or MINIRECORDCORE) structures to be inserted.

You have to be especially careful with the members `pRecordOrder` and `pRecordParent`. At the present stage we are only describing the insertion phase of some objects in a container. The assumption is that the container is empty, because it has just been created. There is much interaction between the user and a window of the `WC_CONTAINER` class. Therefore it is reasonable to assume that the number of records contained in a container can vary. Imagine the addition of a new object as a result of a *drag & drop* operation. In this case you need to repeat each of the operations described above (memory allocation, and filling in the `RECORDCORE` and `RECORDINSERT` structures). If the new object is inserted directly into the container, there is a good chance that it will become a “first-level” entity, that is unbound from all others. The members `pRecordOrder` and `pRecordParent` are thus assigned the following values:

```
...
recins.pRecordOrder = (RECORDCORE)CMA_FIRST ;
recins.pRecordParent = NULL ;
...
```

If, instead, the addition of an object happens together with the release of another object (provided the two objects are compatible), then insertion has to follow another scheme. In order to respect the hierarchical structure (the structure that is shown in the *tree view* mode), it is necessary that `pRecordParent` contain the address of the parent (the object under the mouse when the insertion is the result of a *drag & drop*), and `pRecordOrder` contains the define `CMA_END`:

```
...
recins.pRecordOrder = (RECORDCORE)CMA_END ;
recins.pRecordParent = precParent ;
...
```

Once all these operations are completed, there is nothing else to do other than physically inserting the new records into the container. Chapter 12 will describe this technique in great detail. This task is delegated to `CM_INSERTRECORD`.

<code>CM_INSERTRECORD</code>	0x0339	<i>Description</i>
<code>mp1</code>	<code>RECORDCORE</code> <code>pRecord</code>	Address of a <code>RECORDCORE</code> (or <code>MINIRECORDCORE</code> ) structure
<code>mp2</code>	<code>RECORDINSERT</code> <code>pRecordInsert</code>	Address of a <code>RECORDINSERT</code> structure
Return Value	<code>ULONG</code> <code>cRecords</code>	Number of structures present in the container or 0 in case of error

Table 7.36 lists the attributes of the message `CM_INSERTRECORD`.



**Table 7.36 The Attributes of the Message CM\_INSERTRECORD**

<i>CM_INSERTRECORD Attribute</i>	<i>Value</i>
CMA_TOP	0x0001
CMA_BOTTOM	0x0002
CMA_LEFT	0x0004
CMA_RIGHT	0x0008
CMA_FIRST	0x0010
CMA_LAST	0x0020
CMA_END	0x0040
CMA_PREV	0x0080
CMA_NEXT	0x0100
CMA_HORIZONTAL	0x0200
CMA_VERTICAL	0x0400
CMA_ICON	0x0800
CMA_TEXT	0x1000
CMA_PARTIAL	0x2000
CMA_COMPLETE	0x4000

This is not the whole picture, though! The last step in inserting new data is using the CNRINFO to define some operative and functional aspects of the container.

```
typedef struct _CNRINFO
{ // ccinfo
  ULONG cb ;
  PVOID pSortRecord ;
  PFIELDINFO pFieldInfoLast ;
  PFIELDINFO pFieldInfoObject ;
  PSZ pszCnrTitle ;
  ULONG flWindowAttr ;
  POINTL ptlOrigin ;
  ULONG cDelta ;
  ULONG cRecords ;
  SIZEL slBitmapOrIcon ;
  SIZEL slTreeBitmapOrIcon ;
  HBITMAP hbmExpanded ;
  HBITMAP hbmCollapsed ;
  HPOINTER hpPtrExpanded ;
  HPOINTER hpPtrCollapsed ;
  LONG cyLineSpacing ;
  LONG cxTreeIndent ;
  LONG cxTreeLine ;
  ULONG cFields ;
  LONG xVertSplitbar ;
} CNRINFO ;

typedef CNRINFO *PCNRINFO ;
```

The 19 members of the CNRINFO structure fulfill various needs pertaining to the numerous operating modes of WC\_CONTAINER class windows. Most of the time you will specify only some of the values, and ignore the rest. This is what happens, for instance, at the end of the preparatory phase of the data to be inserted into a container. In the sample listed in Listing 7.11 only a title is provided for the window. The member pszCnrTitle contains the text string to be displayed as the window's title (actually the presence of a title implies the creation of a window belonging to the nondocumented class #52).

A member of CNRINFO that plays a fundamental role is flWindowAttr: It contains the attributes of a container. These are one or more values introduced by the prefix CA\_ (Table 7.37). Returning to the example of the title, it will be necessary to set the attribute CA\_CONTAINERTITLE together with CA\_TITLELEFT, CA\_TITLECENTER, or CA\_TITLERIGHT to define the alignment. Customization of a container's title is completed by the attribute CA\_TITLESEPARATOR that will draw a horizontal separator line between the title area and the actual window area.

```
...
ccinfo.cb = sizeof( CNRINFO) ;
ccinfo.pszCnrTitle = "Container title" ;
ccinfo.flWindowAttr = CA_CONTAINERTITLE | CA_TITLECENTER |
                    CA_TITLESEPARATOR ;
...
```

**Table 7.37 The Attributes of the Class WC\_CONTAINER**

<i>Attribute</i>	<i>Value</i>	<i>Description</i>
CA_CONTAINERTITLE	0x00000200L	A title is added to the container.
CA_TITLESEPARATOR	0x00000400L	A horizontal separator line will be drawn beneath the titlebar.
CA_TITLELEFT	0x00000800L	The title is left-aligned.
CA_TITLERIGHT	0x00001000L	The title is right-aligned.
CA_TITLECENTER	0x00002000L	The title is centered.
CA_OWNERDRAW	0x00004000L	The container is owner-drawn.
CA_DETAILSVIEWTTITLES	0x00008000L	
CA_ORDEREDTARGETEMPH	0x00010000L	
CA_DRAWBITMAP	0x00020000L	The container displays bitmaps.
CA_DRAWICON	0x00040000L	The container displays icons.
CA_TITLEREADONLY	0x00080000L	Prevents the user from changing the container's title.
CA_OWNERPAINTBACKGROUND	0x00100000L	Let the application paint the container background.
CA_MIXEDTARGETEMPH	0x00200000L	
CA_TREELINE	0x00400000L	Draws a line between records in tree view.

**Table 7.38 The Display Modes of a Container**

<i>Mode</i>	<i>Value</i>	<i>Description</i>
CV_TEXT	0x00000001L	Text view
CV_NAME	0x00000002L	Name view
CV_ICON	0x00000004L	Icon view
CV_DETAIL	0x00000008L	Details view
CV_FLOW	0x00000010L	Flow view
CV_MINI	0x00000020L	Mini-icon view
CV_TREE	0x00000040L	Tree view

In this way the container class will examine only the member `pszCnrTitle`, as indicated by the set of attribute in `flWindowAttr` summarized in Table 7.37.

There is yet one more aspect to consider, regarding the display mode of the container. All the preparatory operations performed up to this point assumed that we dealt with *icon view* mode, even if this was never stated explicitly. The member `flWindowAttr` will define this aspect. In addition to the attributes introduced by the prefix `CA_`, there are also others introduced by the prefix `CV_`, listed in Table 7.38.

The selection is rich and full of details, and does not correspond one-to-one with what is offered by WPS in the second level menu introduced by Open. We can now add the define `CV_ICON` to the previous code fragment:

```
...
ccinfo.flWindowAttr = CV_ICON | CA_CONTAINERTITLE |
                      CA_TITLECENTER | CA_TITLESEPARATOR ;
...
```

and then proceed with passing all information by issuing the message `CM_SETCNRINFO`, containing in `mp1` the address of the `CNRINFO` structure, and in `mp2` a set of `CMA_` flags indicating which members of `CNRINFO` are to be considered.

		<i>Description</i>
<code>CM_SETCNRINFO</code>	0x0349	
<code>mp1</code>	<code>PCNRINFO pCnrInfo</code>	Address of a <code>CNRINFO</code> structure
<code>mp2</code>	<code>ULONG ulCnrInfoFl</code>	Set of <code>CMA_</code> flags
Return Value	<code>BOOL fSuccess</code>	Success or failure of the operation

In our example, the whole becomes:

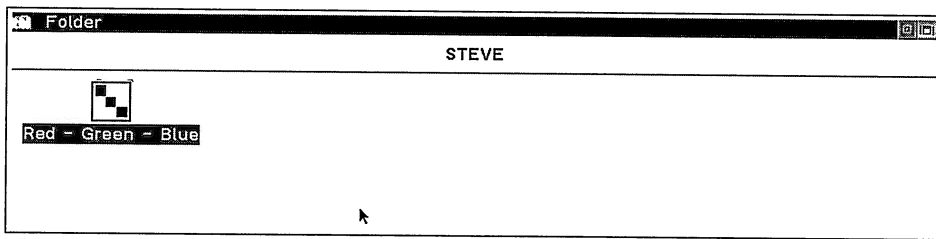
```
...
WinSendMessage( hwndCnr, CM_SETCNRINFO,
                MPFROMP( &ccinfo),
                MPFROMLONG( CMA_CNRTITLE | CMA_FLWINDOWATTR) );
...
```

Table 7.39 lists all attributes of the message `CM_SETCNRINFO`.

Only at this point can we actually start seeing something inside a container, as shown in Figure 7.27.

**Table 7.39** The Attributes of the CM\_SETCNRINFO Message

<i>CM_SETCNRINFO</i> Attributes	<i>Value</i>	<i>Description</i>
CMA_CNRTITLE	0x0001	Container's title.
CMA_DELTA	0x0002	Defines space between adjacent records.
CMA_FLWINDOWATTR	0x0004	Attributes of the container window.
CMA_LINESPACING	0x0008	Vertical line spacing, expressed in pixels, between two adjacent records.
CMA_PFIELDINFOLAST	0x0010	Indicates the presence of the address to the window's left-most column in <i>details view</i> with <i>split windows</i> .
CMA_PSORTRECORD	0x0020	Indicates the presence of a function for sorting the contents of the container.
CMA_PTLORIGIN	0x0040	Origin of the lower left-hand corner in the output space ( <i>workspace</i> ) of a container in <i>icon view</i> mode.
CMA_SLBITMAPORICON	0x0080	Size in pixels of the icon or bitmap displayed in the container.
CMA_XVERTSPLITBAR	0x0100	Initial position of the <i>split bar</i> with respect to the left border.
CMA_PFIELDINFOOBJECT	0x0200	Indicate the presence of the address to the column that identifies the object in <i>details view</i> .
CMA_TREEICON	0x0400	Icons displayed to represent an expanded or collapsed node in the tree structure.
CMA_TREEBITMAP	0x0800	Bitmap displayed to represent an expanded or collapsed node in the tree structure.
CMA_CXTREEINDENT	0x1000	Measurement in pixels of the indentation of branches in the tree structure.
CMA_CXTREELINE	0x2000	Width in pixels of the line that links contiguous branches in the tree structure.
CMA_SLTREEBITMAPORICON	0x4000	Size in pixels of the icon or bitmap displayed in the container to represent the tree structure.

**Figure 7.27** The look of the FOLDER application immediately after being loaded.

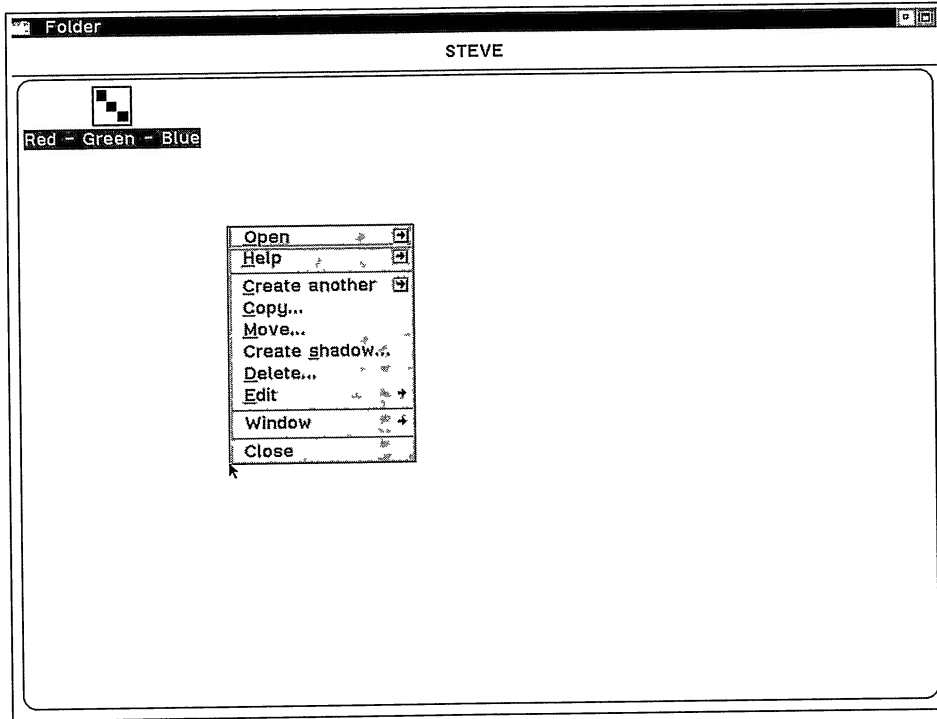


Figure 7.28 The FOLDER application has a window context menu that provides an Open menu item.

## Diversions with Containers

The above title might not be the best way to describe the preliminary and preparatory operations for a container. The look of the FOLDER application depicted in Figure 7.28 is rather poor, especially considering the amount of work done. Inside the container there appears only one object, despite the insertion of four objects: one at root level and three as its children. These last items are invisible because of the *icon view* mode that limits the presentation only to those objects that are lacking a *parent* in the RECORDINSERT structure only (root-level objects). It would be instructive to switch to the *tree view* mode to discover the flexibility and power of containers. Just press the right mouse button over any empty position in the container and the *window context menu* will appear (Figure 7.28).

Select Tree view, and the contents will quickly change its display (Figure 7.29). Now you can see all objects: the parent RGB and the three color icons red, green, and blue.

Switching to the new mode simply requires the issuing of the message CM\_SETCNINFO provided you have filled in appropriately some members of the CNINFO structure.

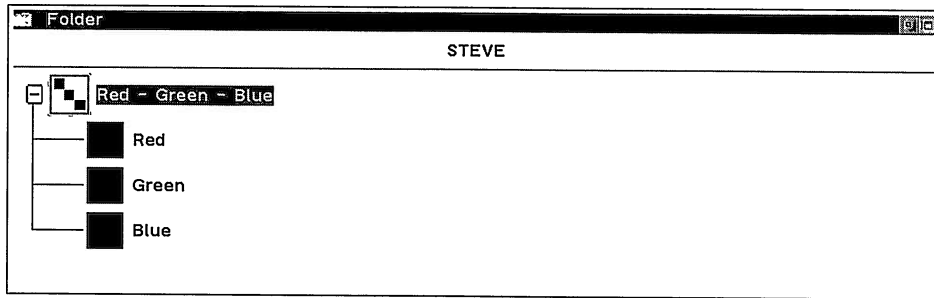


Figure 7.29 The FOLDER application in the Tree view mode.

```

...
// query the container
WinSendMessage( hwnd, CM_QUERYCNRINFO,
                MPFROMP( &cnrinfo),
                MPFROMSHORT( sizeof( CNRINFO)) );

// set the Tree view
cnrinfo.flWindowAttr = CV_TREE | CV_ICON | CA_TREELINE |
                      CA_TITLESEPARATOR | CA_CONTAINERTITLE ;

WinSendMessage( hwnd, CM_SETCNRINFO,
                MPFROMP( &cnrinfo),
                MPFROMLONG( CMA_FLWINDOWATTR | CMA_CNRTITLE)) ;

...

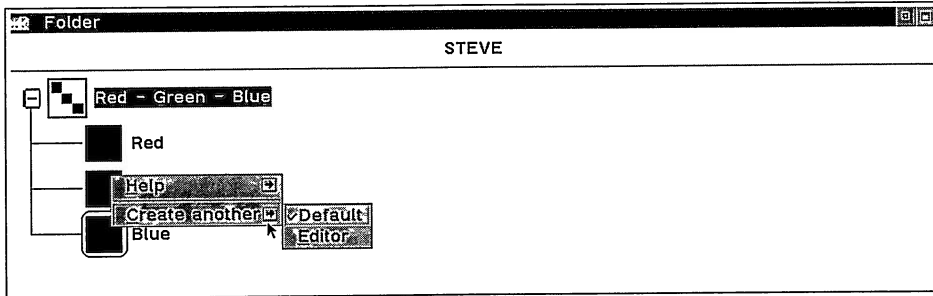
```

The presence of the attributes `CV_TREE` and `CV_ICON` is justified by the possible variations of the tree view mode. The retrieval of information through `CM_QUERYCNRINFO` makes the preparatory phase of the `CNRINFO` structure somewhat simpler. Therefore, neither the *window context menu* nor the other display modes can be activated automatically. It must always be the application's code that takes care of the change.

## The Window Context Menu

The *window context menu* of the container window was created according to the rules described in the previous chapter, in Listing 6.11. In the case of a container, the scenario is slightly more complicated because, in general, even the single items inside the container will have menus of their own. This has been provided for even in the FOLDER application, as you can see in Figure 7.30.

Pressing the right mouse button over an empty position in the window will cause the application's *window context menu* to be displayed. When the same action is performed over an object in the container, the application has to display a different menu. The "activation" area of the objects varies according to the current display mode. For instance, in tree view, the area will even cover the portion of the window



**Figure 7.30** Each object in FOLDER has a window context menu of its own.

that contains the lines linking objects, and is definitely larger than the icon itself (CV\_ICON).

The presence of a menu for each object is compatible with the same operating model of the equivalent menu for the container itself, even if a good deal of its inner working is based on information defined by the application and associated with objects by means of the APPREC structure. The most complicated component to implement is sensing the mouse's cursor in the window. The notification code CN\_CONTEXTMENU contains in mp2 the address of a RECORDCORE structure whenever the right button mouse click happens inside the area occupied by an object. Otherwise it contains NULL if the event is over an empty portion of the container. It is this piece of information that will let the code discriminate between the two possible alternatives.

Related to this problem is another one: that of emphasizing and then displaying the associated popup menu. The programmer can get to know about the rectangle occupied by an object by sending the message CM\_QUERYRECORDRECT, which returns information in a RECTL structure containing values expressed in the container's units.

Particularly interesting in this context is the logic followed for emphasizing. In addition to the mentioned CM\_QUERYRECORDRECT message that allows you to evaluate the size of an object, the application will also resort to CM\_QUERYVIEWPORTRECT to evaluate the size of the container window. Different from what would happen with *WinQueryWindowRect()*, this will automatically exclude the area taken up by the title. The rounded rectangle to emphasize the object possessing the popup menu is displayed by a call to *GpiBox()*, a call that is prepared by some other GPI operation that will optimize the ensuing drawing and clearing phase.

```

...
hps = WinGetPS( CLIENT( hwnd) );
GpiSetMix( hps, FM_INVERT );
GpiSetColor( hps, CLR_PALEGRAY );
GpiSetCurrentPosition( hps, (PPOINTL)&rcEmph + 1 );
hRound = vRound = 30L;
GpiBox( hps, DRO_OUTLINE, (PPOINTL)&rcEmph, hRound, vRound );
...

```

The display of a menu will cause the window to lose focus completely. Apart from any action that the user might perform with the mouse or the keyboard, the container will receive the notification code `CN_SETFOCUS`: This is the best place in the code where you can clear the emphasis previously applied.

## *Proliferation of Objects*

Selecting the option *Create another* will cause the generation of another object of the same type. This behavior is applicable to colored icons spawned at the end of a “branch.” The same action applied to a RGB object will instead produce a new item at the “root” level, followed by the three icons at the lower level. Proliferation of objects is an interesting thing to examine because it involves two important aspects for managing containers:

- Dynamic memory allocation for the new records
- Handling of focus and emphasis

Here we will not implement the addition of new objects by means of *drag & drop*, because that subject will be covered in Chapter 13. The presence of new objects is generated internally after the selection of the *Create another* option. This action triggers a sequence of operations similar to those described for the container’s preparatory phase. Each item will be inserted in a well-defined position in the chain of objects as a consequence of the links between single `RECORDCORE` structures. These structures are tied together in a single-linked list by means of the `preccnextRecord` member. To make the insertion between `APPREC` structures easier, the address of the parent `RECORDCORE` structure of each branch is included (Figure 7.31).

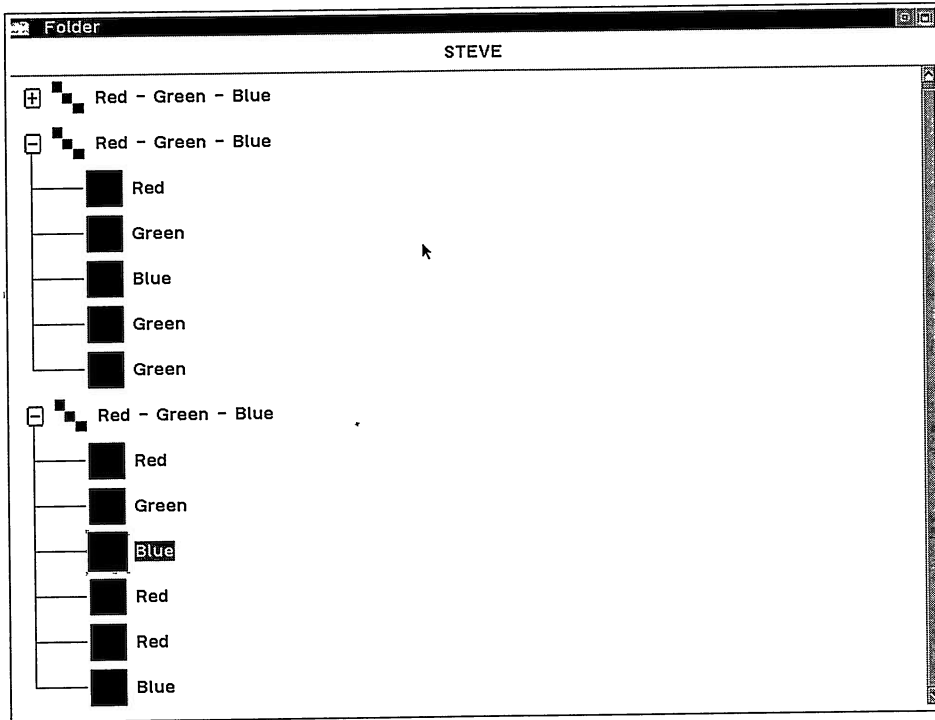
Hence, the proliferation of objects is the direct consequence of a simple manual action, and the use of the same code that inserts a record. It is important to design this portion of the program so as to minimize the impact due of possible user manipulation, which affects the design. To this end, the notification codes of the `WC_CONTAINER` class play a vital role (Table 7.40).

Often, `mp2` contains the address of a `RECORDCORE` (or `MINIRECORDCORE`) structure corresponding to the record on which an action is being performed. For other notification codes, `mp2` is the address of a special data structure that will have among its members a `RECORDCORE`. This is the case, for instance, of `CN_ENTER`; the structure being pointed at by `mp2` is `NOTIFYRECORDENTER`:

```
typedef struct _NOTIFYRECORDENTER
{ // notrecen
    HWND hwndCnr ;
    ULONG fKey ;
    RECORDCORE pRecord ;
} NOTIFYRECORDENTER ;

typedef NOTIFYRECORDENTER *PNOTIFYRECORDENTER ;
```





**Figure 7.31** Population of FOLDER with new objects of the type Red, Green, Blue, and RGB.

In this case, the accessible data are: the container's handle (which is most useful for sending yet other messages), an indicator of the input tool used (keyboard or mouse), and the address of the RECORDCORE structure affected by the action. A double-click with the mouse or depressing the Enter key is detected by the container's owner's window procedure as the notification code `CN_ENTER`. Other data structures complete the syntax of the `WC_CONTAINER` class, as listed in Table 7.41.

In Table 7.42 there is a summary of the use of the `CMA_` attributes; here the attributes are grouped according to the messages to which they refer.

## Other Actions on Containers

One of the key features of WPS is the possibility of direct manipulation of objects, and here containers play a crucial part. As you can gather from the table of notification codes, a container will pass to its owner information pertaining to the interactions that take place between the user and the objects it contains. We will better deal with this aspect of using containers in Chapter 12, after learning about the general rules governing drag & drop. The sample code of FOLDER appears in Listing 7.11.



**Table 7.40 The Notification Codes of the Class WC\_CONTAINER**

<i>Notification Code</i>	<i>Value</i>	<i>Description</i>
CN_DRAGAFTER	101	Sent to the owner after the receipt of a DM_DRAGOVER in the container with CA_ORDEREDTARGETEMPHASIS or CA_MIXEDTARGETEMPHASIS, and with a display mode of Name, Text, or Detail.
CN_DRAGLEAVE	102	Sent to the owner after the receipt of a DM_DRAGLEAVE.
CN_DRAGOVER	103	Sent to the owner after the receipt of a DM_DRAGOVER, and when the container does not have the attribute CA_ORDEREDTARGETEMPHASIS, and it is in the display mode Tree or Icon.
CN_DROP	104	Sent to the owner after the receipt of a DM_DROP.
CN_DROPHELP	105	Sent to the owner after the receipt of a DM_DROPHELP.
CN_ENTER	106	Sent to the owner after the pressing of the Enter key, or after a double-click with the selection button.
CN_INITDRAG	107	Sent to the owner when a <i>dragging</i> operation is initiated.
CN_EMPHASIS	108	Sent to the owner when a record attribute changes in the container.
CN_KILLFOCUS	109	Sent to the owner when the container loses focus.
CN_SCROLL	110	Sent when scrolling takes place.
CN_QUERYDELTA	111	Sent to get additional information when scrolling is being executed.
CN_SETFOCUS	112	Sent to the owner when the container acquires focus.
CN_REALLOCPSZ	113	Sent to the owner before the editing phase of one of the container's items terminates (CN_ENDEDIT).
CN_BEGINEDIT	114	Sent when a text string is being changed in the container.
CN_ENDEDIT	115	Sent to the owner when the editing phase of one of the container's items terminates.
CN_COLLAPSETREE	116	Sent after a tree structure is collapsed in the tree view display mode.
CN_EXPANDTREE	117	Sent after a tree structure is expanded in the tree view display mode.
CN_HELP	118	Sent to the owner after the receipt of a WM_HELP.
CN_CONTEXTMENU	119	Sent to the owner when the container receives the message WM_CONTEXTMENU.

**Table 7.41 Other Data Structures Related to the Usage of Containers**

<i>Structure</i>	<i>Event</i>
CNRDRAGINFO	Drag & drop actions.
NOTIFYRECORDEMPHASIS	Selection and emphasis of an object.
NOTIFYRECORDENTER	Double-click or pressing of the Enter key over an object.
NOTIFYDELTA	Request of the position of an object in the container.
NOTIFYSCROLL	Scrolling actions.
CNREDITDATA	Editing actions.

**Table 7.42 The CMA\_ Attributes and the Messages to Which They Refer**

<i>CM_QUERYRECORD</i>		
<i>Attribute</i>	<i>Value</i>	<i>Description</i>
CMA_PARENT	0x0001	Returns the address of the RECORDCORE structure of the parent record.
CMA_FIRSTCHILD	0x0002	Returns the address of the RECORDCORE structure of the first child record.
CMA_LASTCHILD	0x0004	Returns the address of the RECORDCORE structure of the last child record.
CMA_FIRST	0x0010	Returns the address of the RECORDCORE structure of the first record in the container.
CMA_LAST	0x0020	Returns the address of the RECORDCORE structure of the last record in the container.
CMA_PREV	0x0080	Returns the address of the RECORDCORE structure of the previous record in the container.
CMA_NEXT	0x0100	Returns the address of the RECORDCORE structure of the next record in the container.
<i>CM_QUERYRECORD-FROMRECT</i>		
<i>Attribute</i>	<i>Value</i>	<i>Description</i>
CMA_ITEMORDER	0x0001	Records are listed in numerical order.
CMA_COMPLETE	0x4000	Returns information on records that are completely inside the indicated rectangle.
CMA_PARTIAL	0x2000	Returns information on records that are even partially inside the indicated rectangle.
CMA_ZORDER	0x0008	Records are listed according to their position.
<i>CM_Attribute</i>	<i>Value</i>	<i>Description</i>
CMA_DELTATOP	0x0001	Delta at top of record.
CMA_DELTABOT	0x0002	Delta at bottom of record.
CMA_DELTAHOME	0x0004	Delta at top of list.
CMA_DELTAEND	0x0008	Delta at bottom of list.

*(continued)*

Table 7.42 (Continued)

<i>CM_Attribute</i>	<i>Value</i>	<i>Description</i>
CMA_NOREPOSITION	0x0001	Do not reposition record.
CMA_REPOSITION	0x0002	Reposition record.
CMA_TEXTCHANGED	0x0004	Record text has changed.
CMA_ERASE	0x0008	Record has been erased.
<b>CM_REMOVERECORD</b>		
<i>Attribute</i>	<i>Value</i>	<i>Description</i>
CMA_FREE	0x0001	Record Memory has been erased.
CMA_INVALIDATE	0x0002	Container content is invalidated after a record has been removed.

A last operation is the disposal of allocated memory. The destruction of a record happens either when the user deletes an object (but this operation is not implemented in FOLDER, although it is easy to implement by adding a suitable *menuitem*) or when the program terminates. The message CM\_REMOVERECORD allows you to destroy a record selectively, or to destroy all records present in a container. The following code fragment deals with the complete removal of the contents of a container during the termination phase of an application.

```

...
prec = (RECORDCORE)WinSendMessage( hwnd, CM_QUERYRECORD,
                                   NULL,
                                   MPFROM2SHORT( CMA_FIRST,
                                                  CMA_ITEMORDER) );
if( WinSendMessage( hwnd, CM_REMOVERECORD,
                   MPFROM( prec),
                   MPFROM2SHORT( 0, CMA_FREE)) == 0)
    WinAlarm( HWND_DESKTOP, WA_ERROR );
...

```

Maintaining correct memory management is an absolute priority for the functioning of a container window. In Chapter 13 we will further explore this class when we study the example SNOOPWPS.

## The Class WC\_SLIDER

The class WC\_SLIDER comprises windows that are specialized in the representation of scalar values. The look of a scalar is similar to the volume control of many turntables that have an arm running along a shaft. The *Keyboard* object of the *System Setup* of WPS presents a number of sliders to establish how the keyboard is to respond when keys are pressed (Figure 7.32).

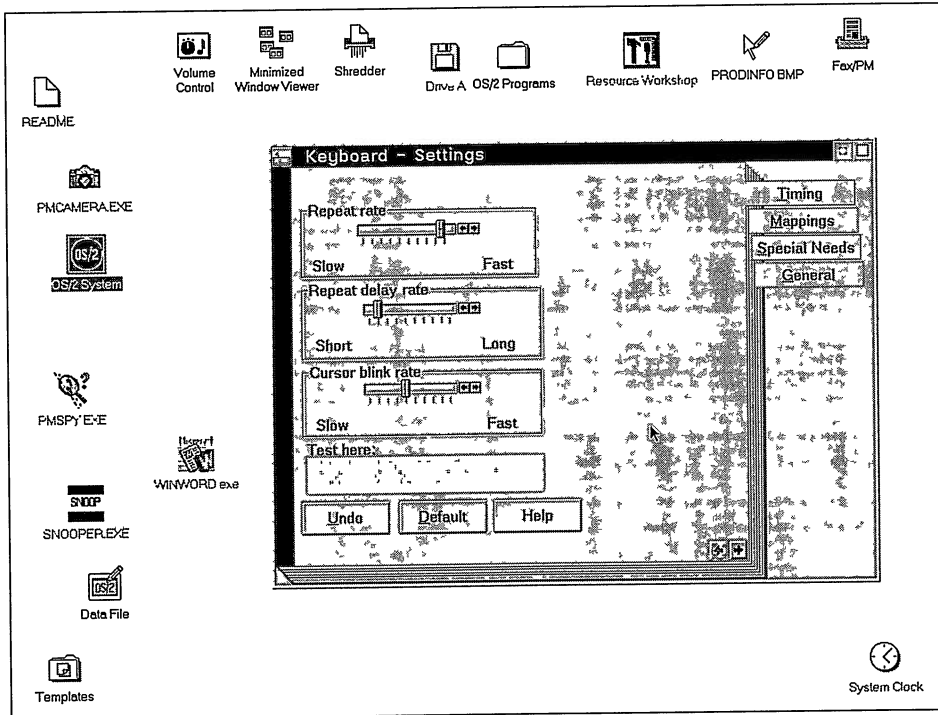


Figure 7.32 Some sliders in the Keyboard object of WPS.

Sliders can be used in a number of ways in applications: defining the amount of primary (RGB) colors when setting a new hue, setting the volume in multimedia applications, fixing the zoom factor in the display of an image, and in many other situations. The number of SLS\_ styles is remarkable, as you can see in Table 7.43.

Table 7.43 The Styles of the Class WC\_SLIDER

Style	Value	Description
SLS_HORIZONTAL	00000000L	Defines a horizontal slider (default).
SLS_VERTICAL	00000001L	Defines a vertical slider.
SLS_CENTER	00000000L	The shaft is central, immediately underneath the slider.
SLS_BOTTOM	00000002L	The shaft is below the slider.
SLS_TOP	00000004L	The shaft is above the slider.
SLS_LEFT	00000002L	The shaft is vertical and to the left of the slider (requires SLS_VERTICAL).
SLS_RIGHT	00000004L	The shaft is vertical and to the right of the slider (requires SLS_VERTICAL).

(continued)

**Table 7.43 (Continued)**

<i>Style</i>	<i>Value</i>	<i>Description</i>
SLS_SNAPTOINCREMENT	00000008L	The position indicator snaps to the nearest tick mark, and is redrawn when it is over a position that is not an integer multiple of the scale.
SLS_BUTTONSBOTTOM	00000010L	Places two buttons under the shaft for controlling the movement of the slider.
SLS_BUTTONSTOP	00000020L	Places two buttons over the shaft for controlling the movement of the slider.
SLS_BUTTONSLEFT	00000010L	Places two buttons to the left of the shaft for controlling the movement of the slider (only for vertical ones).
SLS_BUTTONSRIGHT	00000020L	Places two buttons to the right of the shaft for controlling the movement of the slider (only for vertical ones).
SLS_OWNERDRAW	00000040L	The slider is redrawn by the application.
SLS_READONLY	00000080L	The user is not allowed to interact with the slider (only the application will).
SLS_RIBBONSTRIP	00000100L	The space for the indicator and the starting position is filled with a color different from the ordinary one so as to emphasize any change in a colorful way.
SLS_HOMEBOTTOM	00000000L	The starting position of the slider is at the lower end (default for vertical sliders).
SLS_HOMETOP	00000200L	The starting position of the slider is at the upper end.
SLS_HOMELEFT	00000000L	The starting position of the slider is at the left end (default for horizontal sliders).
SLS_HOMERIGHT	00000200L	The starting position of the slider is at the right end.
SLS_PRIMARYSCALE1	00000000L	The value increments of the slider are determined by the values of the primary scale above the shaft (for a horizontal slider) or to the right of it (for a vertical slider).
SLS_PRIMARYSCALE2	00000400L	The value increments of the slider are determined by the values of the secondary scale below the shaft (for a horizontal slider) or to the left of it (for a vertical slider).

The structural elements that constitute a slider are many. The window will always occupy a rectangle containing the shaft, which is by default much smaller than the whole window (to be precise, it is thinner if it is a horizontal slider and narrower if it is a vertical one). Some styles can be selected according to the layout (horizontal or vertical), which thus must be the first aspect to consider when designing a window of this class.

Along the shaft you will have a proportionally sized arm. The designer can decide the placement of the shaft within the window through the styles SLS\_TOP and SLS\_BOTTOM, while SLS\_CENTER is the default. To move the arm, the user has a couple of buttons that can be placed at both ends of the shaft (above or below, to the right or to the left, respectively with SLS\_BUTTONSTOP or SLS\_BUTTONSBOTTOM, and SLS\_BUTTONSRIGHT or SLS\_BUTTONSLEFT). Furthermore, the programmer can indicate the direction of movement of the arm (with the style SLS\_HOMExxx). The list is completed with the styles that affect the structural elements of a slider: SLS\_PRIMARYSCALE1 and SLS\_PRIMARYSCALE2. The effect of these two styles is to define how the arm can move along the shaft. When the arm departs from its starting position, the shaft is divided into two portions. Assuming the arm is moving from left to right, the left part of the arm is controlled by the style SLS\_RIBBONSTRIP, while the right part is identified by the shaft itself, with no corresponding style (Figure 7.33).

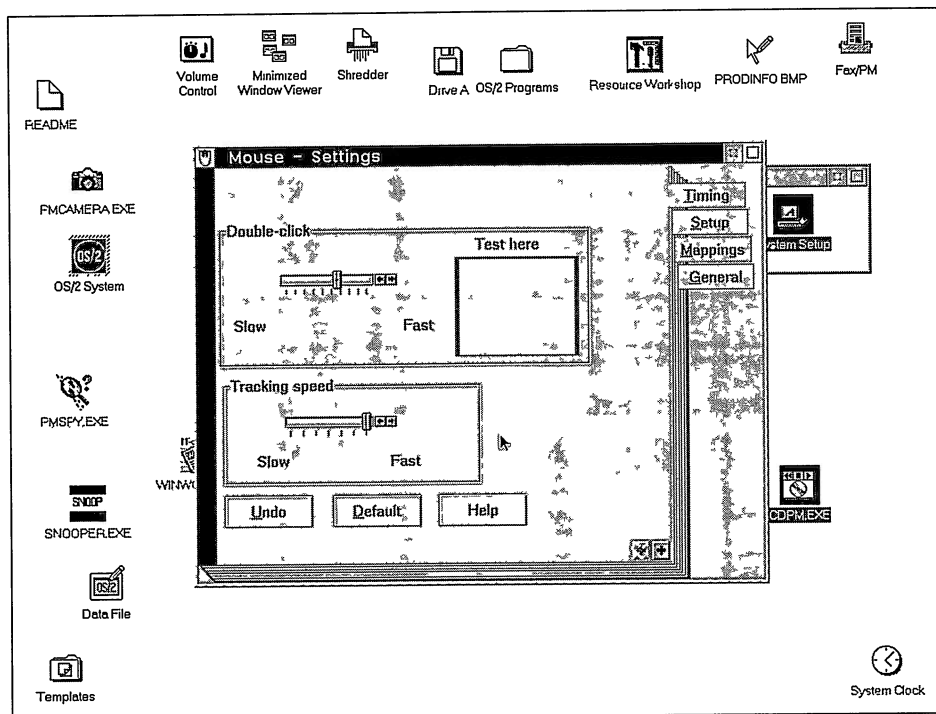


Figure 7.33 The structural elements of a slider.

With SLS\_RIBBONSTRIP the part of the shaft between the origin and the arm will be colored dark gray. Unfortunately, it is not possible to define the color as you like; you can only accept CLR\_DARKGRAY instead of CLR\_PALEGRAY which is used to color the remaining part of a slider. Furthermore, there happens to be an erroneous repaint action of a slider equipped with this flag when the window is resized.

The selection of a value is not only the consequence of the user's action on the arrows of the slider. As with scrollbars, in this case it is possible to act directly upon the arm. If the flag SLS\_SNAPTOINCREMENT is set for the slider, then the movement along the shaft will occur only by fixed amounts, corresponding to each tick mark along the shaft.

The flag SLS\_READONLY will prevent the user from interacting directly with the slider. A practical consequence of this is the disappearance of the arm as well as the movement arrows. This is the only way to achieve this despite the presence of SLM\_SETSLIDERINFO dedicated, among other things, to setting its size. The last flag of the class WC\_SLIDER is SLS\_OWNERDRAW, which will implement an object whose output is controlled by its owner.

## *Creating a Slider*

As usual, the class WC\_SLIDER has a special data structure for creating a window instance:

```
typedef struct _SLDCDATA
{ // sldcd
    ULONG cbSize ;
    USHORT usScale1Increments ;
    USHORT usScale1Spacing ;
    USHORT usScale2Increments ;
    USHORT usScale2Spacing ;
} SLDCDATA ;

typedef SLDCDATA *PSLDCDATA ;
```

In addition to indicating the overall size of the structure, the programmer will usually indicate the number of tick marks requested for the primary and secondary scales. The space between two consecutive tick marks is computed automatically by the class by indicating the value of 0 for the two members usScale1Spacing and usScale2Spacing.

## *Messages of the Class WC\_SLIDER*

The appearance of tick marks to indicate the possible positions that the slider can take, and the appearance of any text associated with each position, is a consequence of the use of the class's messages (Table 7.44). The simplest way to display tick marks is to send the message SLM\_SETTICKSIZE, and specify the attribute SMA\_SETALLTICKS (Table 7.46).



**Table 7.44 The Messages of the Class WC\_SLIDER**

<i>Message</i>	<i>Value</i>	<i>Description</i>
SLM_ADDDETENT	0x0369	Adds indications to the shaft.
SLM_QUERYDETENTPOS	0x036a	Returns the position of the current indicator.
SLM_QUERYSCALETEXT	0x036b	Returns the text associated with a primary scale tick mark.
SLM_QUERYSLIDERINFO	0x036c	Returns the positions and sizes of the components of a slider.
SLM_QUERYTICKPOS	0x036d	Returns the current position of a detent along the primary scale.
SLM_QUERYTICKSIZE	0x036e	Returns the size of a detent along the primary scale.
SLM_REMOVEDETENT	0x036f	Removes a detent.
SLM_SETSCALETEXT	0x0370	Displays text over a detent.
SLM_SETSLIDERINFO	0x0371	Defines the slider information.
SLM_SETTICKSIZE	0x0372	Defines the size of a detent along the primary scale.

	<i>Value</i>	<i>Description</i>
SLM_SETTICKSIZE	0x0372	
mp1	USHORT usTickNum USHORT usTickSize	Tick(s) to be activated Size of the tick mark
mp2	ULONG reserved	Reserved
Return Value	BOOL fResult	Success or failure

```
...
WinSendMessage( hwndSlider, SLM_SETTICKSIZE,
                MPFROM2SHORT( SMA_SETALLTICKS, 10), 0L );
...
```

The second SHORT in mp1 corresponds to the height in pixels of the tick marks on the screen. The descriptive text of each tick mark will always appear inside the overall parameters of the WC\_SLIDER class window, thanks to the SLM\_SETSCALETEXT message.

	<i>Value</i>	<i>Description</i>
SLM_SETSCALETEXT	0x0370	
mp1	USHORT usTickNum	Tick(s) to be activated
mp2	PSZ pszTickText	Text to be displayed
Return Value	BOOL fResult	Success or failure

The list of messages also includes SLM\_ADDDETENT, which will identify along the shaft all unnecessary positions that overlap a tick. Imagine, for instance, that you need to represent the number of rounds per minute of an engine by means of a

**Table 7.45 The Notification Codes of the Class WC\_SLIDER**

<i>Notification Code</i>	<i>Value</i>	<i>Description</i>
SLN_CHANGE	1	Movement of the position indicator.
SLN_SLIDERTRACK	2	Dragging of the position indicator.
SLN_SETFOCUS	3	Acquisition of focus.
SLN_KILLFOCUS	4	Loss of focus.

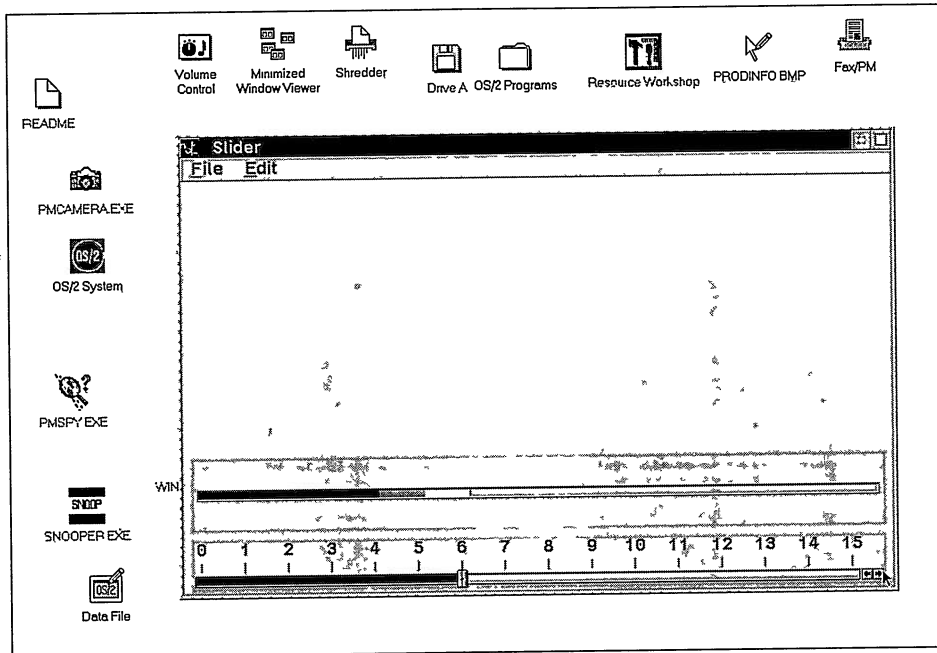
vertical slider with an upper limit of 7,000. Notwithstanding the maximum value of the slider, 5,000 rpms are to be considered an alarming position: It is the ideal placement for indicating a *detent*.

The information returned to the owner of a WC\_SLIDER class window is rather limited (Table 7.45). The notification codes are always forwarded through WM\_CONTROL.

Above all, it is only SLN\_CHANGE that informs the application of the movement of the arm along the shaft, and indicates the new position in pixels. This often means that you have to perform conversions in terms of tick marks, which is an unpleasant operation because it requires you to know the overall length of the slider and the exact number of tick marks it contains. It is easier to send the message SLM\_QUERYSLIDER-INFO, and specify the pair of attributes SMA\_SLIDERARMPOSITION and SMA\_INCREMENTVALUE to get this information (Table 7.46). Furthermore, it would be desirable to have some means for inferring the direction of movement without having to keep the previous location stored.

**Table 7.46 The Attributes of the Messages of the Class WC\_SLIDER**

<i>Attribute</i>	<i>Value</i>	<i>Description</i>
SMA_SCALE1	0x0001	Indicator's position on primary scale.
SMA_SCALE2	0x0002	Indicator's position on secondary scale.
SMA_SHAFTDIMENSIONS	0x0000	Size of the shaft.
SMA_SHAFTPOSITION	0x0001	Position of the lower right-hand corner of the shaft.
SMA_SLIDERARMDIMENSIONS	0x0002	Size of the slider's arm.
SMA_SLIDERARMPOSITION	0x0003	Position of the slider's arm.
SMA_RANGEVALUE	0x0000	Pixels from the starting position to the current position of the arm.
SMA_INCREMENTVALUE	0x0001	Increment value on the primary scale.
SMA_SETALLTICKS	0xFFFF	Sets all the tick marks to the size specified.



**Figure 7.34** The SLIDER application contains a standard slider and a owner-draw slider.

## A Sample Slider



Figure 7.34 depicts the SLIDER application (listing 7.12) that uses two sliders inside the client window. By acting upon the arrows of the lower slider you can change the client's color. Any movement will cause a corresponding movement even in the upper slider, which is a read-only slider and cannot be acted on by the user. This action is governed internally by the programmer by sending the message `SLM_SETSLIDERINFO`. The upper slider is of the *owner-draw* type and is also provided with a colored shaft.

A *owner-draw slider* receives the message `WM_DRAWITEM` when it is necessary to generate some output. The address of the structure `OWNERITEM` is always contained in `mp1`. Many members of this structure cannot be used because they are reserved. The application's must integrate the few pieces of information passed by the `WC_SLIDER` with values computed in the code. The portion of the window that needs repainting is indicated by the defines listed in Table 7.47.

---

## The Class WC\_SPINBUTTON

A spinbutton is a kind of cross between an entryfield and a miniature vertical scrollbar. The result is an ideal window for controlled input of a numeric value, or even text strings. The style of the class is introduced by the prefix `SPBS_` (Table 7.48).

**Table 7.47 The Values of the idItem Member of the OWNERDRAW Structure of a Slider Created with the SLS\_OWNERDRAW Style**

<i>Attribute</i>	<i>Value</i>	<i>Description</i>
SDA_RIBBONSTRIP	0x0001	Redraw the area occupied by the shaft.
SDA_SLIDERSHAFT	0x0002	Redraw only the shaft rail.
SDA_BACKGROUND	0x0003	Redraw the position indicator of the slider.
SDA_SLIDERARM	0x0004	Redraw the background of the slider.

The outfit of styles of this class will affect mostly the display mode of the contents of the spinbutton. A very interesting flag is SPBS\_PADWITHZERO which will pad a number with leading zeros in order to make it a predetermined length. Setting SPBS\_MASTER will entail the presence of a pair of vertical arrows used to scroll the contents of the spinbutton (this kind of window is very common in the *Settings* of WPS). The adoption of SPBS\_READONLY disables any input functionality in the entry-field portion, and offers the designer a useful tool when the options presented to the user need to be completely controlled by the application. The messages of the class (Table 7.49) mostly affect input of data into the window.

**Table 7.48 The Styles of the Class WC\_SPINBUTTON**

<i>Style</i>	<i>Value</i>	<i>Description</i>
SPBS_ALLCHARACTERS	0x00000000L	Accepts any character (default).
SPBS_NUMERICONLY	0x00000001L	Accepts only numbers.
SPBS_READONLY	0x00000002L	The entryfield window is for output only.
SPBS_MASTER	0x00000010L	The spinbutton is equipped with a miniature scrollbar.
SPBS_SERVANT	0x00000000L	The spinbutton is not equipped with a miniature scrollbar.
SPBS_JUSTDEFAULT	0x00000000L	Default justification (left).
SPBS_JUSTLEFT	0x00000008L	Left alignment.
SPBS_JUSTRIGHT	0x00000004L	Right alignment.
SPBS_JUSTCENTER	0x0000000CL	Centered.
SPBS_NOBORDER	0x00000020L	Window without border.
SPBS_FASTSPIN	0x00000100L	Fast spinning of numbers that don't have to appear in sequential order.
SPBS_PADWITHZEROS	0x00000080L	The number is padded with zeros.

**Table 7.49 The Messages of the Class WC\_SPINBUTTON**

<i>Message</i>	<i>Value</i>	<i>Description</i>
SPBM_OVERRIDESTLIMITS	0x200	Changes the limits of scrolling without affecting the current value.
SPBM_QUERYLIMITS	0x201	Returns the limits set via SPBM_SETLIMITS.
SPBM_SETTEXTLIMIT	0x202	Defines the maximum number of characters that are allowed in the entryfield.
SPBM_SPINUP	0x203	Selects the next available value.
SPBM_SPINDOWN	0x204	Selects the previous available value.
SPBM_QUERYVALUE	0x205	Returns the current value.
SPBM_SETARRAY	0x206	Changes the displayed data to be displayed by specifying noncontiguous values.
SPBM_SETLIMITS	0x207	Defines the upper and lower limits of scrolling.
SPBM_SETCURRENTVALUE	0x208	Sets the current value.
SPBM_SETMASTER	0x209	Defines the owner of the spinbutton.

A spinbutton will display a sequence of numbers or a series of values or text strings previously defined. In the case of scrolling numbers, it is sufficient to indicate the upper and lower limits through SPBM\_SETLIMITS by indicating in mp1 and mp2, respectively, the upper and the lower limits.

	<i>Value</i>	<i>Description</i>
SPBM_SETLIMITS	0x0207	
mp1	LONG IUpLimit	Upper numeric limit
mp2	LONG ILowLimit	Lower numeric limit
Return Value	BOOL fResult	Success or failure

In addition to indicating the extreme values, with SPBM\_SETCURRENTVALUE the application can also set the current value displayed in the spinbutton.

	<i>Value</i>	<i>Description</i>
SPBM_SETCURRENTVALUE	0x0208	
mp1	LONG IValue	Value to be displayed
mp2	ULONG reserved	Reserved
Return Value	BOOL fResult	Success or failure

To insert nonsequential numeric values or alphanumeric strings, it is necessary to follow a different path. The insertion is delegated to the message SPBM\_SETARRAY, although SPBM\_SETCURRENTVALUE will always be used to indicate the current value in the window.

		<i>Description</i>
SPBM_SETARRAY	0x0206	
mp1	PSZ pszStr1	Address of an array of pointer
mp2	USHORT usItems	Number of elements in the array
Return Value	BOOL fResult	Success or failure

In mp1 you must specify the name of an array of pointers to char, i.e., a contiguous sequence of memory addresses that refers to the text string that will be displayed in the spinbutton. For instance, imagine you need to display the three RGB colors (Red, Green, and Blue). This becomes:

```
...
CHAR *szString[] = { "Red", "Green", "Blue" } ;
...
WinSendMessage( hwndSpin, SPBM_SETARRAY,
                MPFROMP( szString), MPFROMSHORT( 3) ) ;
...
```

The definition of the current value is always executed through SPBM\_SETCURRENTVALUE, though in mp1 you will not indicate the value to be displayed, but the index into the array to the pointer that needs to be dereferenced (in the example, this will result in the string Green).

```
...
WinSendMessage( hwndSpin, SPBM_SETCURRENTVALUE, MPFROMLONG( 1L), 0L) ;
...
```

 In Listing 7.13 you can see the program SPIN (Figure 7.35) that illustrates the use of this class of windows.

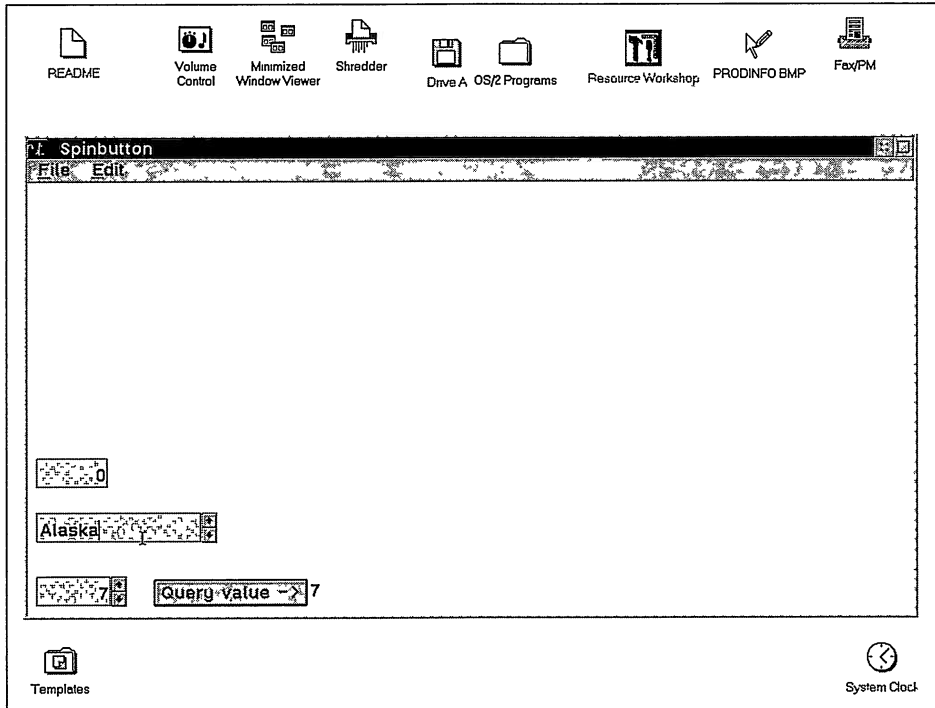
If you search for a WC\_SPINBUTTON class window with SNOOPER you will have a tough time. In fact, a spinbutton is the sum of an entryfield and a window belonging to the undocumented class #57. The window of the class WC\_SPINBUTTON is actually behind these two windows, invisible to the user. The entryfield portion will always have the same ID as the spinbutton, so it is a simple matter to get to its handle:

```
...
u1ID = (ULONG)WinQueryWindowUShort( hwndSpin, QWS_ID) ;
hwndEntry = WinWindowFromID( hwndSpin, u1ID) ;
...
```

The same approach is used for the two vertical scrolling arrows, considering that in this case the ID is always equal to 0L:

```
...
hwndArrows = WinWindowFromID( hwndSpin, 0L) ;
...
```

It is unlikely that you will ever need these two handles, but this is nonetheless an efficient approach, at least at the present stage (it would be good if IBM's designers assigned a constant ID to the two portions of a spinbutton).



**Figure 7.35** When the SPIN program is executing it shows a numeric master spinbutton in the lower part, and a servant spinbutton in the upper part containing the names of all states visited by the author during 1992.

## Master or Servant?

By default, each spinbutton is always of the servant type; that means it will not have the two scroll arrows. Their appearance is induced by setting the style `SBPS_MASTER`. Apart from this, to perform the scrolling of the values available in a spinbutton you need only transfer focus onto the window (a simple mouse click), and then press the up or down arrow cursor key. Naturally, the presence of the two scroll arrows will greatly simplify the interaction with a spinbutton, especially if you are using a mouse. The API of this class presents the message `SPBM_SETMASTER` to allow scrolling a spinbutton without scroll arrows by acting on the arrows of another spinbutton of the master type. The outcome is simplicity itself:

```
...
WinSendMessage( hwndSpinServant, SBPM_SETMASTER,
                MPFROMHND( hwndSpinMaster), NULL) ;
...
```

Just act on the arrows of the master and you will achieve scrolling of this spinbutton. The documentation in the Toolkit is most misleading here. In fact, it seems that you need to get the handle of the servant's entryfield portion, and then pass the handle to

the master. Not true! And the designer will be at a loss. In this case, the problem is not in the API, but in the interface itself. In order to have the master do the scrolling for the servant, it is necessary that the focus be on the servant. Only if the focus is on the servant will it be possible for it to interact with the master. In Listing 7.13 the numeric spinbuttons are linked one to the other by a master-server relationship.

## The Notification Codes

The notification codes of the class WC\_SPINBUTTON are easy to understand and don't require any special explanation (Table 7.50). The code SPBN\_CHANGE, as its name suggests, refers to any possible changes that take place in the entryfield portion of the window.

## A Sample Spinbutton

Listing 7.13 (the application SPIN) also shows how to retrieve the current value of a spinbutton by means of the SPBM\_QUERYVALUE message. The syntax of SPBM\_QUERYVALUE is dependent on the contents of the spinbutton (number or text string):

SPBM_QUERYVALUE	0x205	<i>Description</i>
mp1	PVOID storage	Address of a LONG to hold the number or of a memory area to contain the text string
mp2	USHORT usBufSize	Size of the buffer or 0 if mp1 is the address of a LONG
	USHORT usValue	One of the flags listed in Table 7.51
Return Value	BOOL fResult	Success or failure

In general, it is convenient to specify in `usValue` the define `SPBQ_UPDATEIFVALID`, especially for those spinbuttons that allow the user to input. This flag does not allow the display of a value inconsistent with what is established in the scrolling range.

**Table 7.50 The Notification Codes of the Class WC\_SPINBUTTON**

<i>Notification Code</i>	<i>Value</i>	<i>Description</i>
SPBN_UPARROW	0x20A	Press of the up arrow key.
SPBN_DOWNARROW	0x20B	Press of the down arrow key.
SPBN_ENDSPIN	0x20C	Termination of spinning.
SPBN_CHANGE	0x20D	The contents of the entryfield portion have been changed.
SPBN_SETFOCUS	0x20E	Acquisition of focus.
SPBN_KILLFOCUS	0x20F	Loss of focus.



**Table 7.51** The Query Flags of the Class WC\_SPINBUTTON

<i>Query Flag</i>	<i>Value</i>	<i>Description</i>
SPBQ_UPDATEIFVALID	1	Updates only if the value is valid
SPBQ_ALWAYSUPDATE	2	Always updates
SPBQ_DONOTUPDATE	3	Never updates

It is interesting to assess in SPIN the logic followed to fill in the spinbutton containing the text string. The resource file lists all states visited by the author in 1992. Their loading is performed, as usual, by *WinLoadString()*. Before proceeding with this repeated operation, however, the code allocates a memory page. This amount is greater than the overall length of all names, but represents the minimum amount of memory that can be allocated in OS/2 2.1. Of the 4096 bytes available, only the first 52 are actually used to store the addresses of the 13 text strings present in STRINGTABLE (13 x 4 = 52). The names of the states are allocated one after the other, starting with the 52nd byte, as illustrated in Figure 7.36.

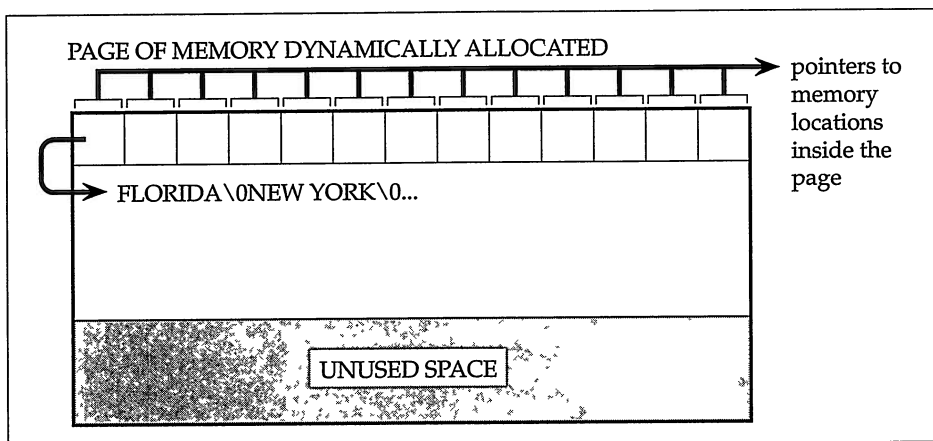
When SPBM\_SETARRAY is issued, in mp1 we indicate the address of the block, and in mp2 the number of states:

```

...
WinSendMessage( hwndSpin, SPBM_SETARRAY,
                MPFROMP( pchStart), MPFROMSHORT( STATES) );
...

```

This scheme is probably the best solution whenever the information to insert in a spinbutton becomes available only during the application's execution.



**Figure 7.36** Scheme adopted for inserting text strings in alphanumeric spinbutton.

**Table 7.52 The Styles of the Class WC\_VALUESET**

<i>Style</i>	<i>Value</i>	<i>Description</i>
VS_BITMAP	0x0001	Contains bitmaps
VS_ICON	0x0002	Contains icons
VS_TEXT	0x0004	Contains text
VS_RGB	0x0008	Contains color data
VS_COLORINDEX	0x0010	Contains color indexes
VS_BORDER	0x0020	Adds a border
VS_ITEMBORDER	0x0040	Adds an outline border around each item
VS_SCALEBITMAPS	0x0080	Scales bitmaps within cells
VS_RIGHTTOLEFT	0x0100	Orders from right to left
VS_OWNERDRAW	0x0200	Owner drawn window

## The Class WC\_VALUESET

The class WC\_VALUESET is a very simple tool, and very useful because of its high degree of flexibility. It is a kind of rectangular container inside which can appear text strings, icons, bitmaps, defines, or colored indexes. A valueset is a kind of control panel with a number of lines and columns defined by the programmer. The objects *Color Palette* and *Font Palette* are very similar to a valueset.

The styles VS\_ (Table 7.52) define the overall features of each item in a valueset. The first five identify the contents of the window. As we will see here, and in the sample VALUESET application of Listing 7.14, the programmer can create a window containing objects that belong to the five kinds supported by the window. Some cells can contain text, others bitmaps, and yet others a color!



With VS\_BORDER the valueset will have a border, while VS\_ITEMBORDER will surround each single item with a thin outline (the two flags can be combined). By default, the insertion of information into the single cells of a valueset will take place from left to right, but this can be changed with VS\_RIGHTTOLEFT. Like most of the more mature classes, even a WC\_VALUESET allows you to generate *owner-draw* windows (VS\_OWNERDRAW), and is able to automatically scale bitmaps to make them fit within a cell.

### Creating a Valueset

Before calling *WinCreateWindow()* it is necessary to declare an identifier of the type VSCDATA, a specific data structure of this class.

```
typedef struct _VSCDATA
{ // vscd
    ULONG cbSize ;
    USHORT usRowCount ;
    USHORT usColumnCount ;
} VSCDATA ;

typedef VSCDATA *PVSCDATA ;
```

In addition to the size of the data structure, the programmer only has to indicate the number of columns and rows that will appear in the valueset. There is no other way to define these fundamental aspects that allow the window to size each cell along both axes.

Almost invariably, when creating a valueset you will set the `VS_` style that identifies the kinds of objects contained in it. According to this approach, the programmer will be limited to inserting information in the cells one at a time. An alternative solution is that of avoiding, during creation, defining the contents of the valueset, and defer the whole process to when values will actually be inserted. The message `VM_SETITEMATTR` will let you specify the contents of each cell, one at a time. This is an approach that can be employed to create a valueset with “mixed contents.”

<code>VM_SETITEMATTR</code>	0x037b	<i>Description</i>
<code>mp1</code>	USHORT usRow	Row
	USHORT usColumn	Column
<code>mp2</code>	USHORT usItemAttr	Attribute of the cell
	USHORT fSet	The attribute is being set (TRUE) or removed (FALSE)
Return Value	BOOL fSuccess	Success or failure

The attributes of the defines introduced by the `VIA_` prefix are listed in Table 7.53.

**Table 7.53 The Attributes of the Messages `VM_SETITEMATTR` and `VM_QUERYITEMATTR`**

<i>Attribute</i>	<i>Value</i>	<i>Description</i>
<code>VIA_BITMAP</code>	0x0001	The cell contains a bitmap.
<code>VIA_ICON</code>	0x0002	The cell contains an icon.
<code>VIA_TEXT</code>	0x0004	The cell contains text.
<code>VIA_RGB</code>	0x0008	The cell contains the numeric description of a color.
<code>VIA_COLORINDEX</code>	0x0010	The cell contains the numeric code ( <code>CLR_</code> ) of a color.
<code>VIA_OWNERDRAW</code>	0x0020	The cell’s output is delegated to the owner.
<code>VIA_DISABLED</code>	0x0040	The cell is disabled.
<code>VIA_DRAGGABLE</code>	0x0080	The contents of the cell can be dragged.
<code>VIA_DROPONABLE</code>	0x0100	The cell can accept an object dragged and dropped into it with the mouse.

**Table 7.54 The Messages of the Class WC\_VALUESET**

<i>Message</i>	<i>Value</i>	<i>Description</i>
VM_QUERYITEM	0x0375	Returns the contents of the item indicated by the row-column pair.
VM_QUERYITEMATTR	0x0376	Returns the attributes of the item indicated by the row-column pair.
VM_QUERYMETRICS	0x0377	Returns the size of each single item or of the space between contiguous items.
VM_QUERYSELECTEDITEM	0x0378	Returns the coordinates of the currently selected item.
VM_SELECTITEM	0x0379	Selects an item.
VM_SETITEM	0x037a	Defines the kind of contents of an item.
VM_SETITEMATTR	0x037b	Set the attributes of an item.
VM_SETMETRICS	0x037c	Sets the size of an item and the space between adjacent items, or both values.

The attributes `VIA_DROPONABLE` and `VIA_DRAGGABLE` concern drag & drop operations that will be described in Chapter 12. It is interesting to observe that `VIA_OWNERDRAW` assigns the owner-draw feature selectively to one or more cells of a valueset. By taking advantage of this functionality and of the drag & drop conventions, you could easily implement very powerful windows with high-level user interaction.

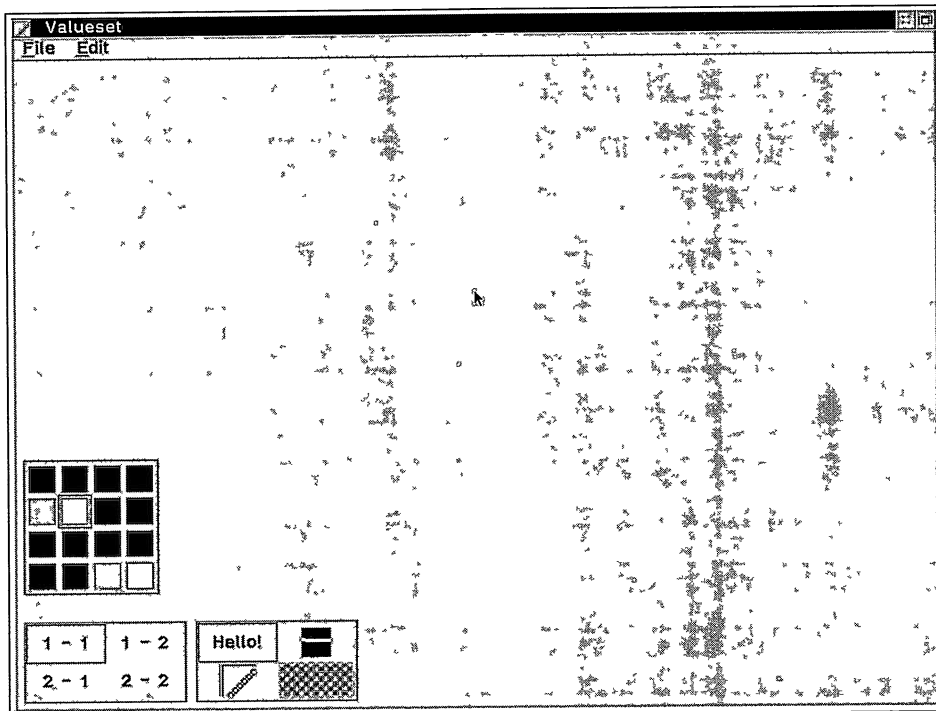
The `VM_` messages will then allow you to get to the single item (`VM_QUERYSELECTEDITEM`), know its contents (`VM_QUERYITEM`) and attributes (`VM_QUERYITEMATTR`), as listed in Table 7.54.

**Table 7.55 The Notification Codes of the Class WC\_VALUESET**

<i>Notification Code</i>	<i>Value</i>	<i>Description</i>
VN_SELECT	120	Selects an item.
VN_ENTER	121	Selects and confirms an item
VN_DRAGLEAVE	122	Receipt of the message <code>DM_DRAGLEAVE</code> .
VN_DRAGOVER	123	Receipt of the message <code>DM_DRAGOVER</code> .
VN_DROP	124	Receipt of the message <code>DM_DROP</code> .
VN_DROPHELP	125	Receipt of the message <code>DM_DROPHELP</code> .
VN_INITDRAG	126	Start of a drag operation.
VN_SETFOCUS	127	Acquirement of focus.
VN_KILLFOCUS	128	Loss of focus.
VN_HELP	129	Receipt of the message <code>WM_HELP</code> .

The selection through the mouse of an item will cause the notification code `VN_SELECT` to be sent by means of the `WM_CONTROL` message. A double-click or pressing the Enter key will be detected as `VN_ENTER`. Other notification codes mostly concern drag & drop operations, as illustrated in Table 7.55.

In Figure 7.37 you can see the application `VALUESET`; it contains three windows of this class. The first window, on the lower left side, contains a piece of text, and helps in understanding the default order followed when identifying cells. The valueset to the right is an example of “mixed contents”: there are cells containing text, bitmaps, icons, and a randomly selected color. The third and last valueset displays the 16 available colors corresponding to the `CLR_` defines in `PMGPI.H`. The selection of an item will cause the client to be colored in that hue.



**Figure 7.37** The output of `VALUESET` illustrates the different kinds of windows of this class.



# Dialog Windows

After looking through so many windows, we will now have a look at . . . another window! In this chapter we will examine *dialog windows*. A dialog window (or simply a *dialog*) is similar to the windows we created earlier with `WinCreateStdWindow()` or `WinCreateWindow()` with some additional features. In the first place, dialogs play a special role in PM applications: They are the kind of window that appears after selecting an *extended command* from the menu bar (a menu item followed by an ellipsis). It is very unusual to use these windows as an application's main window. This first point must be considered carefully. An extended command implies a much higher degree of interaction between the user and the application before the operation associated with the selected menu item will be performed. Think about the window that appears after selecting an extended command as a kind of extension of the selected menu item. This window must capture the application's input focus exclusively, so that the user must complete the operation requested through the selected menu item. Any attempt to perform an action outside the dialog will be prevented by the system, which will emit a beep. This is already significantly different from an ordinary window. The principal purpose of a dialog is to contain a number of *controls* that allow the user to interact with the application in a flexible and articulate way.

Other peculiarities of dialogs pertain to programming aspects. There is no predefined class for creating a dialog, but it is *not* necessary to register any window class in order to create one. This makes dialogs akin to a predefined window class. However, different from a predefined class, the creation of a dialog holds the programmer responsible for writing the window procedure that will receive its messages (as this is a *dialog*, the procedure is better known as a *dialog procedure*). The user must also use a specific API for creating and destroying the dialog, and for handling the flow of messages.

The most distinctive trait of a dialog is in the process used to create it. The programmer can draw the dialog on the screen by using specific tools available in the Toolkit or with the compilers, save the work, and generate a dialog template in the resource file. In the C source code the only operation to perform is that of loading the dialog template.

## Two Types of Dialogs

OS/2 is equipped with two kinds of dialogs, called *modal* and *modeless* (Figures 8.1 and 8.2). This terminology was introduced by MS Windows in the early 1980s and is explained by the following line of reasoning. In a statistical distribution the modal value corresponds to a high frequency, while the modeless value is much less frequent. In fact, in PM modal dialogs are used much more often than modeless dialogs. It might be expected that modeless dialogs will eventually disappear altogether from applications, since their usefulness is severely limited and replaceable by child windows. Thus, the study of dialogs will in practice focus on modal dialogs. Basically, a dialog is a *frame window*.

### Features of a Dialog

The primary purpose of a dialog is to make it easier for the user to interact with the application, and to exchange information between the user and the application. The screen area occupied by a dialog will contain controls, that is, windows belonging to PM's predefined classes. SNOOPER will allow you to discover that the background of a dialog is a window belonging to the class WC\_FRAME (Figure 8.3).

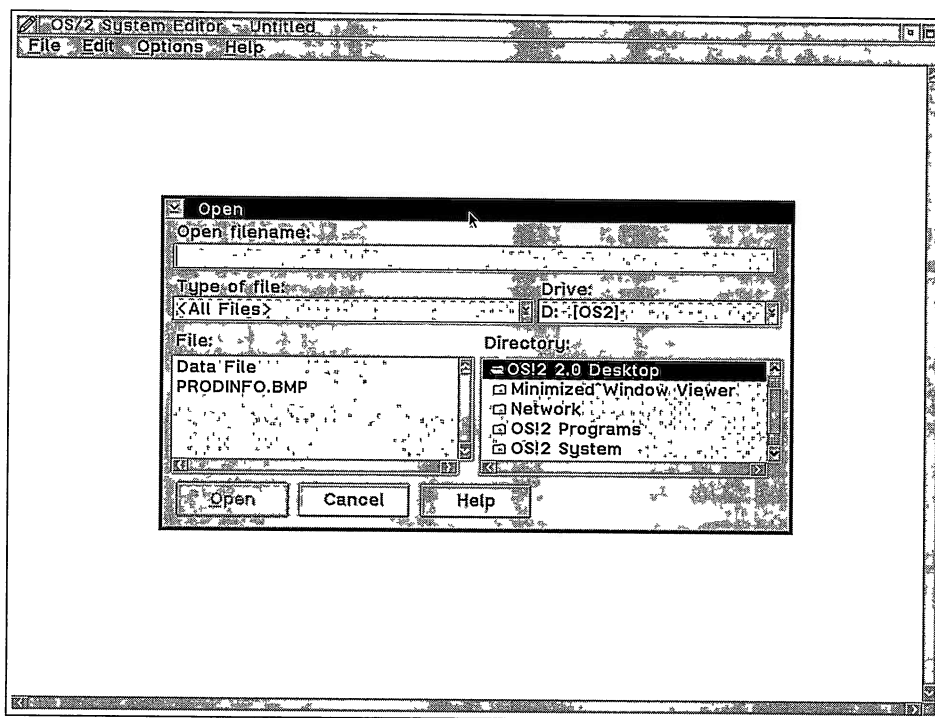


Figure 8.1 A typical modal dialog for loading a file from disk.



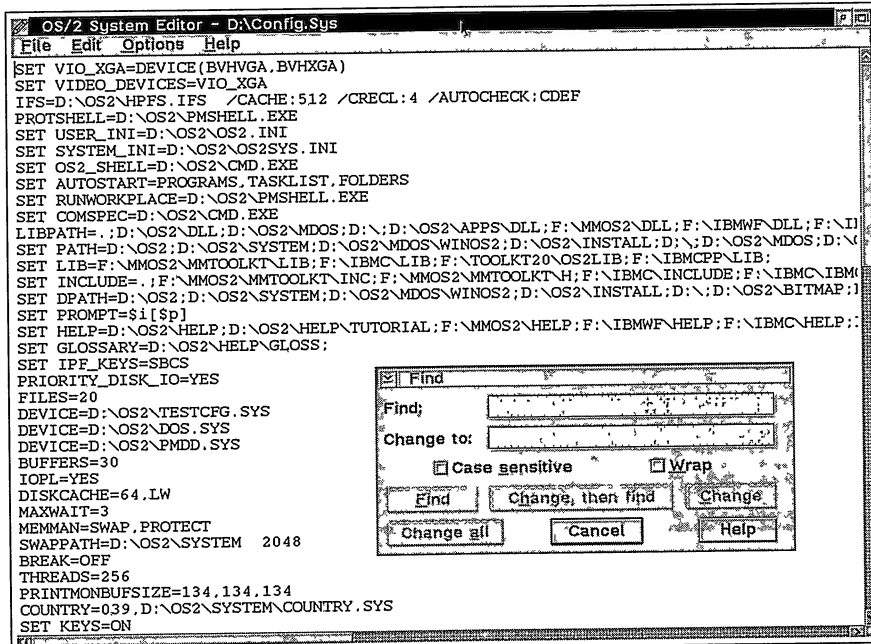


Figure 8.2 A modeless dialog that looks like an ordinary window created with `WinCreateStdWindow()`.

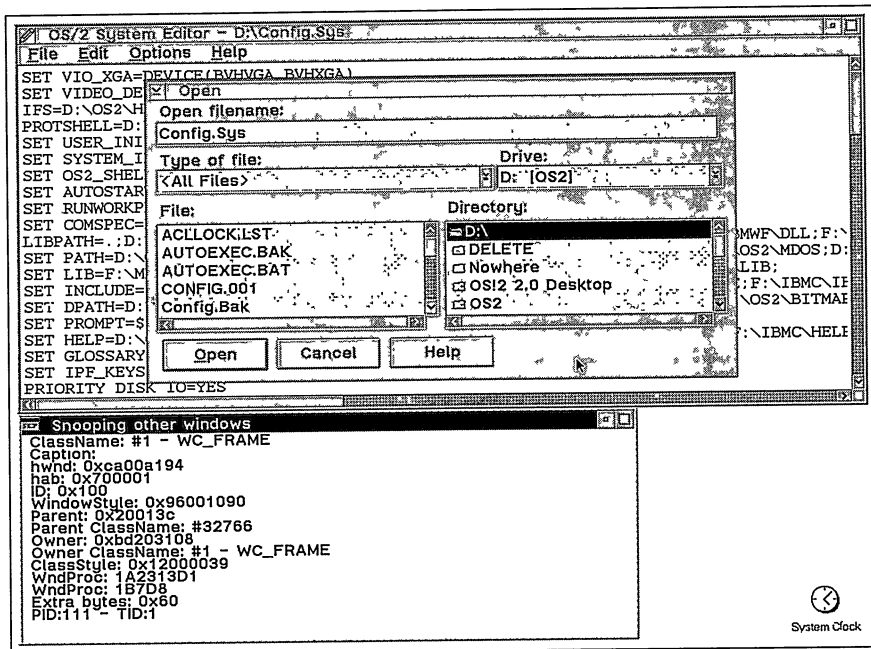


Figure 8.3 The whole area occupied on the screen by a dialog is a window of the class `WC_FRAME`.

The *frame window* is the parent of all controls present, and provides them with the needed pixels. Compared to applications developed in earlier chapters, there is no client window interposed between the frame and the controls, as in most of the samples of Chapter 7.

The visual feature that is most evident in a dialog is the absence of a sizing border. No design rules will prevent you from assigning this attribute to a dialog; but there are other considerations that make it unwise. If a dialog were sizable, then a user might make the controls disappear from view. The direct consequence would be a limitation in the window's intended functionality of being an interaction area with the user.

The border of a dialog is always a thin separator line, or sometimes a thicker one. The first style is `FS_BORDER` and the second one is `FS_DLGBORDER` (Figure 8.4). Then there are the equivalent of *frame control flags*, `FCF_BORDER`, and `FCF_DLGBORDER`.

Often, a dialog will have a titlebar that will allow you to move it around on the screen (Figure 8.5). To the left of the titlebar is the icon of the titlebar menu, containing the options that are traditionally listed under the secondary Window menu.

## Creating a Dialog

A dialog is mainly a container for controls. In Chapter 7 we used the function `WinLoadDlg()` to load the *window template* of a window defined in the resource file. The

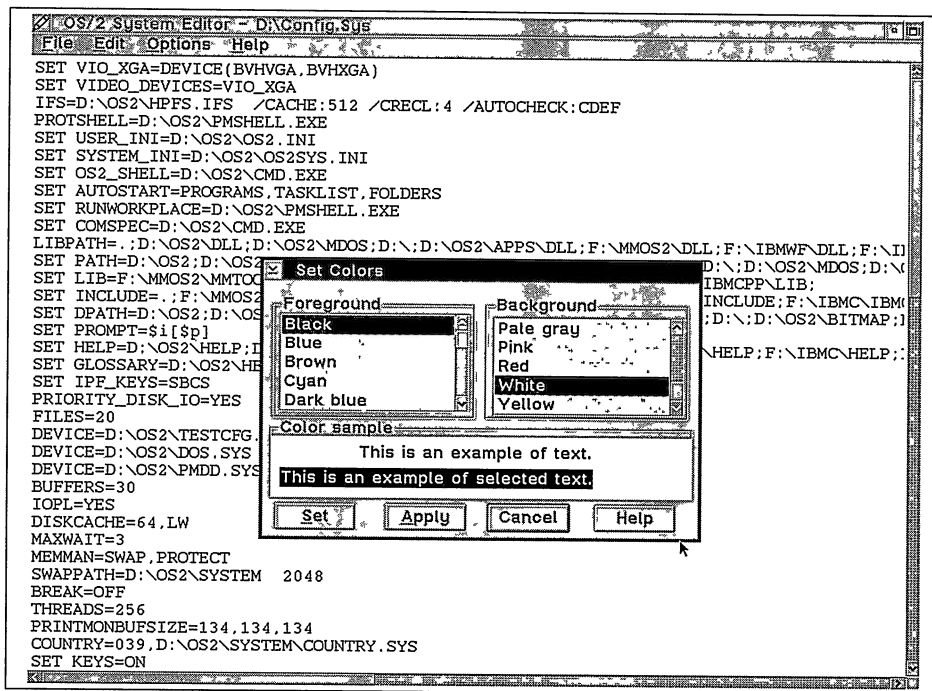


Figure 8.4 A dialog equipped with the typical border due to the style `FS_DLGBORDER`.

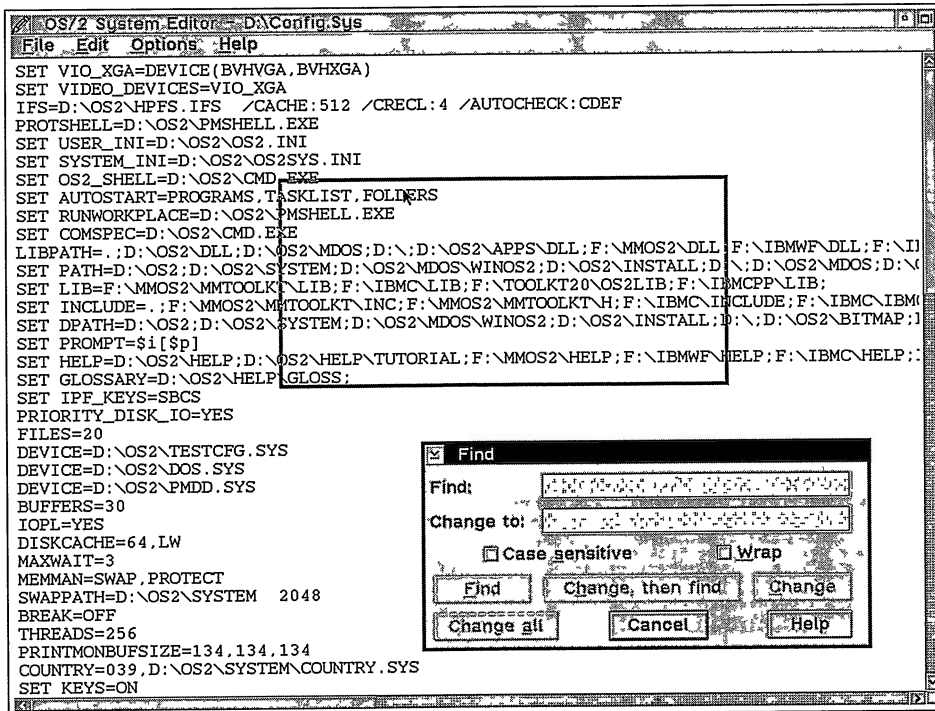


Figure 8.5 A dialog equipped with a titlebar can be moved around on the screen.

name of the function suggests creating a dialog. In fact, *WinLoadDlg()* and *WinDlgBox()* are the principal tools of the PM's API to implement a dialog. They are specialized, respectively, for modeless and modal dialogs. Therefore, the first operative distinction between the two kinds of dialogs is in the API call used to generate them:

```
#define INCL_WINDIALOGS
HWND APIENTRY WinLoadDlg( HWND hwndParent,
                          HWND hwndOwner,
                          PFNWP pfnDlgProc,
                          HMODULE hmod,
                          ULONG idDlg,
                          PVOID pCreateParams) ;
```

Parameter	Description
hwndParent	Handle of the parent window
hwndOwner	Handle of the owner window
pfnDlgProc	Address of the dialog procedure
hmod	Handle of the module from which the dialog template is to be loaded
pCreateParams	Pointer to a memory area containing useful data for the application

<i>Return Value</i>	<i>Description</i>
HWND	Handle of the dialog window or DID_ERROR

```
#define INCL_WINDIALOGS
ULONG APIENTRY WinDlgBox( HWND hwndParent,
                          HWND hwndOwner,
                          PFNWP pfnDlgProc,
                          HMODULE hmod,
                          ULONG idDlg,
                          PVOID pCreateParams) ;
```

<i>Parameter</i>	<i>Description</i>
hwndParent	Handle of the parent window
hwndOwner	Handle of the owner window
pfnDlgProc	Address of the dialog procedure
hmod	Handle of the module from which the dialog template is to be loaded
pCreateParams	Pointer to a memory area containing useful data for the application

<i>Return Value</i>	<i>Description</i>
ULONG	Termination value for dialog procedures

Both of these functions require the same six parameters to load a modeless or a modal dialog. The first HWND defines the dialog's parent window. You can specify any valid handle, although it will usually be the desktop window that will provide the pixels for creating the dialog. This is why dialogs appear almost everywhere on the screen. The owner of the dialog will usually be the client window of the application.

The appearance of a dialog is always the consequence of the user selecting an extended menu command among the available menu items. The message WM\_CCOMMAND will reach the window procedure of the client, and from there a function is called to create the dialog.

The PFNWP parameter is a pointer to the function that will perform as the dialog procedure. It is always a function with a structure similar to the following:

```
MRESULT EXPENTRY DlgProc( HWND hwnd,
                          ULONG msg,
                          MPARAM mp1,
                          MPARAM mp2) ;

{
    switch( msg)
    {
        ...
    }
    return WinDefDlgProc( hwnd, msg, mp1, mp2) ;
}
```

which is almost identical to a generic window procedure, with the exception that the default message processing is provided by *WinDefDlgProc()* instead of *WinDefWindowProc()*.

The handle to the module identifies the source from which the dialog template is to be loaded—the description of the dialog in the resource file. As this is the same as the executable module, you can specify the value of *NULLHANDLE*. Each dialog template in a resource file is characterized by a unique ID that allows it to describe more than one dialog structure for each application. When the dialog is loaded, the correct template must be identified through its corresponding ID.

The last parameter is a *PVOID* pointer. Often, this item is considered as an alien, and is assigned the value of *NULL* because it is not known how to use it. However, when we examine the dialog procedures, we will see how to take advantage of this piece of data.

## *Modal or Modeless?*

The assumptions for creating a dialog are identical, whether it be modal or modeless, so which should you choose and why? The two functions *WinLoadDlg()* and *WinDlgBox()* are identical except for the return value. The first function returns the handle of the newly created window, the frame. *WinDlgBox()* returns instead a simple *ULONG* corresponding to what was indicated in the dialog procedure concerning the pressing of OK or the Cancel button. The preclusion from interacting with the application's other windows whenever a dialog is displayed holds only for modal dialogs, while the modeless ones are almost in all respects ordinary windows that can be managed through a handle. It is this consideration that makes modal dialogs preferable for implementing the code concerned with an *extended command*, and to use a standard window in place of a modeless dialog for other routine operations.

A modeless dialog will treat its dialog procedure in a way comparable to a simple window belonging to a class registered by the application. Different from a standard child window, though, a modeless dialog can drift out of the space of the application window because its parent is the desktop. Notwithstanding the style model adopted (MDI or SDI), modeless dialogs do not offer the programmer any significant advantage apart from the ease of design through specific tools. Listing 7.6 implementing the FLOW application demonstrated that it is possible to create a window belonging to a class registered by the application starting from a *window template* in the resource file. This approach can be extended to include the creation of complex windows containing in its client area several controls, without having to resort to a dialog. Modeless dialogs are interesting solutions only in very special cases, and it is most often preferable to use windows produced with *WinCreateStdWindow()*.

With the advent of version 2.x, there has been a decline of interest even in modal dialogs. The new development rules for the user interface based on the WPS model partially banish the presence of menus, and thus the item that introduces dialogs: the menu item. This does not mean, however, that dialogs have lost their significance. With the decline of menus has come the advance of *notebooks*, featuring several pages

containing a number of controls. Often the best way to insert controls in a page is to rely on a dialog, especially because of the ease with which the set of windows can be produced.

A final consideration is the presence of predefined dialogs, which are preassembled structures ready to perform specific tasks within an application, like opening a file, saving it, searching for a text string, or replacing it. Let's start, though, by examining dialog templates.

## Dialog Templates

The first action item for a dialog is to prepare a dialog template in the resource file; the template will be loaded later, during the application's execution. The Development Toolkit provides a specific utility, called DLGEDIT.EXE, which specializes in creating dialog templates (Figure 8.6).

We will not describe here how to use DLGEDIT, since the program consists of drawing a dialog on the screen, wherein you position controls belonging to predefined classes or even to classes specifically created by the program. Once these design operations are over, the result is saved by the program in an ASCII file with the extension .DLG. In this *script file*, DLGEDIT inserts all resource compiler directives

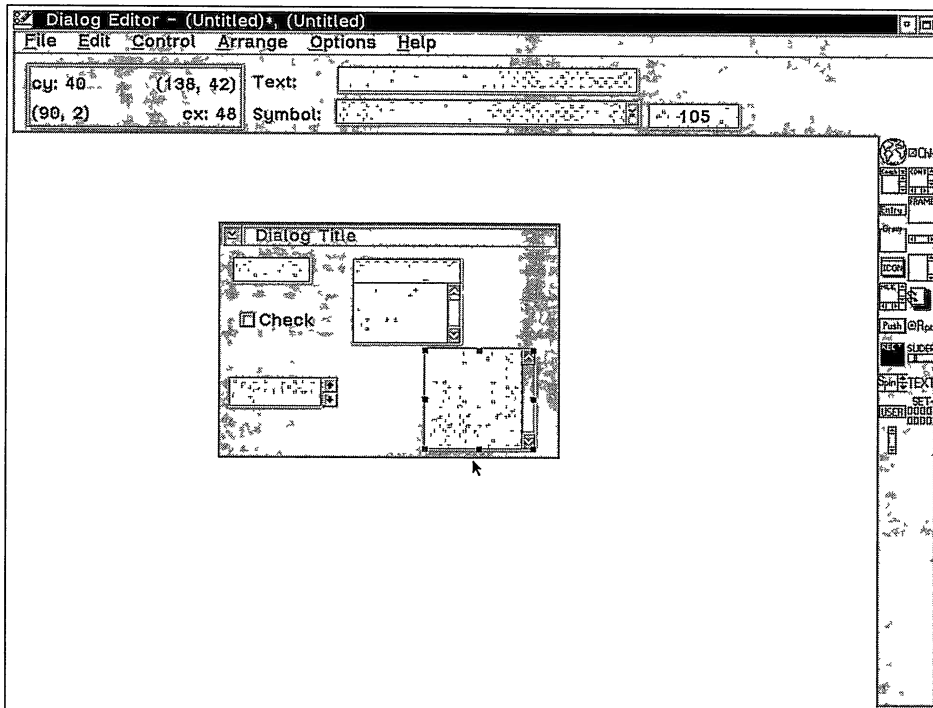


Figure 8.6 With DLGEDIT.EXE, creating a dialog template means drawing several windows on the screen.

necessary for creating the dialog designed on the screen. DLGEDIT also produces a compiled version of the DLG file, assigning it the extension .RES. Figure 8.7 shows a typical dialog template as generated by DLGEDIT.

Nothing prevents the programmer from generating the dialog template manually, provided the specific syntax that requires the use of the DLGTEMPLATE directive under which subordinate blocks have to be inserted for describing the actual dialog and the controls that are present in it are followed.

```
DLGTEMPLATE idtemplate [load options][memory options]
{
    DIALOG text, id, x, y, cx, cy [, [style][,framect1]] [ data def]
    {
        CONTROL text, id, x, y, cx, cy, class [, style] [ data-def]
        ...
    }
}
```

A DLGTEMPLATE resource is identified by a numeric ID. The value to indicate after the DLGTEMPLATE and DIALOG directives must be the same, and it must fall within the range 0-65,536. This value will also be specified as the fifth parameter of either the *WinLoadDlg()* or the *WinDlgBox()* function call. As we have seen in Chapter 7, the syntax of both the DIALOG as well as the CONTROL directives are akin to the parameters of *WinCreateWindow()*, and this is no surprise since the final purpose is the same: creating a window. The automatic generation of the dialog template will often mean that the programmer will rely completely on DLGEDIT for setting the creation and

```
DLGTEMPLATE DL_PRES LOADONCALL MOVEABLE DISCARDABLE
{
    DIALOG          "Presentation params", DL_PRES, 42, 10, 155, 111,
                    FS_NOBYTEALIGN |
                    WS_VISIBLE, FCF_SYSMENU | FCF_TIT-
TLEBAR
    {
        PUSHBUTTON          "Enable", DL_ENABLE, 9,
12, 38, 13, WS_DISABLED
        PUSHBUTTON          "Disable", DL_DISABLE,
51, 12, 38, 13
        DEFPUSHBUTTON       "~OK", DID_OK, 113, 30, 38, 13
        PUSHBUTTON          "~Cancel", MN_FILE, 113,
12, 38, 13
        LISTBOX              DL_LIST, 9, 55,
142, 43, LS_MULTIPLESEL |
WS_GROUP
```

Figure 8.7 A dialog template generated by DLGEDIT.EXE.

display flags of the components of a dialog. The styles employed are the traditional ones: `WS_`, `FS_`, and the frame control flags `FCF_`.

Each control is identified by an ID defined during design phase. A sound approach for large-sized projects is to generate a specific header file listing all defines related to the drawn controls. This new header file is then included in the application's header file, and thus it is accessible both from the `.RC` as well as from the source code. A convention adopted in this text is that of introducing the defines of the control's IDs with the prefix `DL_`. According to this plan, a listbox designed to display the contents of a directory could be indicated with the ID `DL_DIR`, while another one could be indicated for listing the drives and directories with `DL_DRIVES`.

```
...
// dialog header file
#define DL_DIR      100
#define DL_DRIVES  101
...
```

A very common approach is generating a dialog template with `DLGEDIT` or another equivalent tool, and then adjusting the size or the position of controls manually by changing the template. When writing an application, it is convenient to maintain dialog templates separate from the other resources, and leave them in one or more `.DLG` files. The insertion into `.RC` is governed by the `rcinclude` directive, or by the more general `#include` directive; the two have slightly different syntaxes:

```
...
rcinclude mydlg.dlg
...
#include "mydlg.dlg"
...
```

## The Styles `WS_GROUP` and `WS_TABSTOP`

The styles `WS_GROUP` and `WS_TABSTOP`, while they are generic `WS_` styles, are applied almost exclusively in the template of a dialog. The style `WS_GROUP` allows you to collect several controls into one logical group, inside of which you can transfer focus and select items by pressing the cursor control keys. The means by which this is done are somewhat odd. The style `WS_GROUP` must be specified only for the *first* control belonging to the new group. All controls that belong to that group, and that physically follow the define of the first one in the template, must appear without this style. To understand where a group ends, and where a new one starts, you must apply once again the style `WS_GROUP`: The preceding control will be the last one of the first group, and the one with the `WS_GROUP` style will be the first one of the second group. Odd, indeed!

`WS_TABSTOP` is conceptually much simpler. This flag is assigned to each control which the user is allowed to access to by pressing the `TAB` key on the keyboard.



## The Dialog Procedure

A *dialog procedure* is a function similar in most aspects to a window procedure, with only a few slight differences. As with a window procedure, it is mandatory to implement default processing for any message that will be received by the procedure. In this case, you can resort to the specific function called *WinDefDlgProc()*, which specializes in handling the messages of a dialog.

```
#define INCL_WINDIALOGS
MRESULT WinDefDlgProc( HWND hwndDlg,
                       ULONG msg,
                       MPARAM mp1,
                       MPARAM mp2) ;
```

<i>Parameter</i>	<i>Description</i>
hwndDlg	Handle of the dialog
msg	Message
mp1	Parameter
mp2	Parameter
<i>Return Value</i>	<i>Description</i>
MRESULT	Return value of the dialog procedure

All messages in a dialog procedure refer to the possible interactions between the user and the dialog, which is a window belonging to the class `WC_FRAME`. Actually the user will perform only a limited number of actions within the areas of the frame that are occupied by controls. The dialog is simply a pixel provider window for the display of its controls, and thus the user's actions are concentrated on controls. These will tell the dialog what is happening by means of notification codes that are sent to the window procedures of the window belonging to predefined classes. One message that is practically always detected in a dialog procedure is `WM_COMMAND`, as is `WM_CONTROL`. In practice, a dialog procedure can be reduced to processing only these two messages together with a third one, `WM_INITDLG`.

## A Doubtful Question

We started examining dialogs by saying that a dialog procedure is comparable to a window procedure. There are some further points to consider. A dialog is a window belonging to the class `WC_FRAME`. A frame will send its messages to the window procedure of its class, which is certainly a function that has nothing to do with the dialog procedure written by the programmer. This also applies to controls. These are almost invariably windows belonging to predefined classes, and very seldom to a class registered by the application. In both cases, however, the messages produced by the user's actions are always addressed to the window procedure of the corresponding classes. So here's the question: What is a dialog procedure?

It is clear that the application that creates a dialog is not provided with any mechanism for intercepting the message flow generated by the user's interacting with

the window. It is totally excluded, with no apparent means of intervention. The ownership/parenthood relationship, in fact, is with regard to the dialog and its controls. For these windows, the interaction mechanism is notification codes, in addition to specific messages for the various classes. So, in order to let an application understand what actions the user performs in a dialog, it is necessary to supply a kind of bridge into the application code that will allow the dialog to tap into the program's code. Hence, the message flow that runs through the window procedure of the class WC\_FRAME will be partly branched off toward the dialog procedure whenever the involved frame is the parent of a dialog (Figure 8.8).

What goes on here is a kind of extrusion of messages from the window procedure of the class WC\_FRAME toward an appropriate piece of code that will always take the form of a function capable of processing those messages. The abandoning of the dialog procedure means that the message is rendered back to the window procedure of the class WC\_FRAME, where it will be subject to standard processing. This behavior explains the presence of *WinDefDlgProc()* in place of *WinDefWindowProc()*. The action performed by *WinDefDlgProc()* is limited to transferring control to the dialog's handler.

Therefore, messages that reach a dialog procedure are in some way "deviated" and "selected" messages. More precisely, a dialog procedure will receive all messages that are actually addressed to the dialog's frame window. However, the designer will be interested only in the two messages mentioned earlier, because they are the ones that

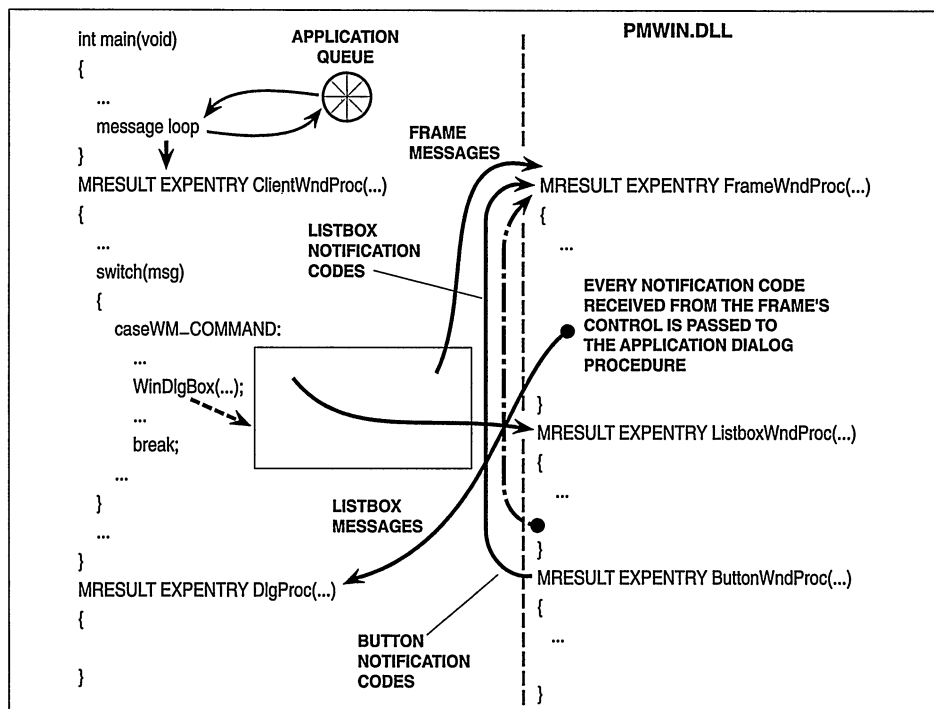


Figure 8.8 The message flow in a dialog.

indicate what actions were executed by the user on the dialog's controls, in addition to WM\_INITDLG, needed for initialization purposes.

---

## The Message WM\_INITDLG

When you design a dialog, you need to decide the position and size of each control it will contain. Even more important, you have to assign to each control a specific functionality that it has to deliver during runtime. In an *open-box*, as in Figure 8.1, you will have a listbox to display the names of drives and directories present in the system, and a second one to contain the file corresponding to what is set in an entryfield. All these data items need to appear in the controls at the moment the dialog is displayed—not before, since it would be impossible, and not later, since it would only confuse the user.

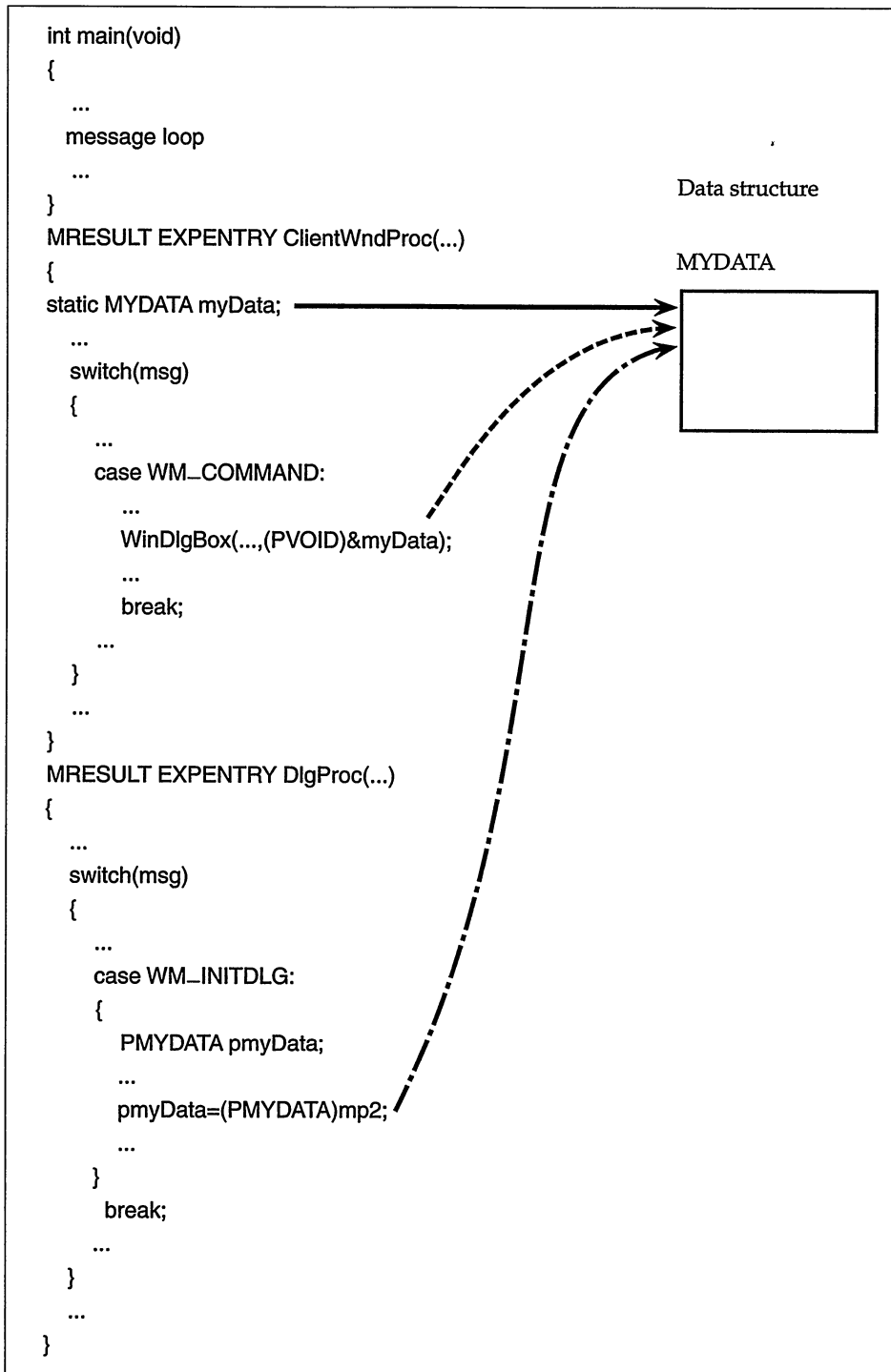
The message WM\_INITDLG reaches the dialog procedure immediately after the frame and all its controls have been created, but before PM displays the dialog. This means that when the application receives the WM\_INITDLG message, all the handles will be available as will all necessary tools for interacting with each single item so that it can be set up to present the appropriate information.

WM_INITDLG	0x003b	<i>Description</i>
mp1	HWND hwnd	Handle of the control that has the input focus
mp2	PCREATEPARAMS- pCreateParams	Pointer to a memory area containing dialog specific data
Return Value	BOOL fResult	TRUE if the window having focus has changed; FALSE if it has not changed

WM\_INITDLG contains interesting pieces of information to customize the look of controls and to take advantage of the dialog.

The message presents in mp1 the handle of the first control, defined at the template level, which has the input focus. The contents of mp2 are much more useful: It is a pointer to a memory area containing application-specific data. By looking again at the syntax of both *WinLoadDlg()* and *WinDlgBox()*, you will notice that the last parameter is a generic pointer to a memory area containing data defined by the programmer. On the basis of the relationship that exists between a dialog procedure and the application, it is obvious that there is a communication problem between these two portions of the code. To avoid resorting to global variables (the worst solution), the dialog's API provides a very efficient mechanism for passing information to a dialog procedure. The final pointer in a *WinLoadDlg()* or a *WinDlgBox()* call, provided it is not a NULL pointer, indicates a memory area allocated by the programmer within which you can insert all data that needs to be passed to the dialog procedure. In this function, access to this information is granted by the value in mp2. Figure 8.9 illustrates the whole mechanism of information passing between the two procedures.

The advantage of this solution lies in having a permanent connecting bridge between the window procedure and the dialog procedure so that the exchange of data can take place in both directions, not just from the caller to the callee. Thus, a dialog



**Figure 8.9** Passing of data between the window procedure and the dialog procedure by taking advantage of the mechanism provided by PM's API.

is capable of informing its owner window about the selections and operations performed by the user. This is done by inserting appropriate data in the shared communication area between the two functions. Once processing returns to the window procedure, the implicated pointer to the private area is still valid, and can be used for reading that information.

Take advantage of this provision by passing all information needed by the dialog procedure at the very moment it is called, rather than using global variables. We will put this into practice in the next few samples.

## The Ownership Problem

When a dialog is created, you have to indicate its parent (generally `HWND_DESKTOP`) and its owner (the *client window* of the main window). Therefore, it would be natural to expect that when you ask a dialog for the handle of its owner, you would be returned the value of the client window. But this is not the case. Instead, the value returned by `WinQueryWindow()` when it is asked for the owner, is a handle different from the client window specified in `WinDlgBox()`.

This strange behavior is not a bug in PM, but a design choice made by IBM in the early stages of designing the API. The SNOOPER utility will help us to understand what is going on. In Figure 8.10, you can see SNOOPER presenting information about the client area of a window that will later generate a dialog.

So, let's now proceed and display a simple modal dialog by selecting the option Product information from the File menu. Figure 8.11 presents the data regarding the dialog as it is retrieved by SNOOPER. The dialog was created starting with the following portion of code:

```
...
case MN_PRODINFO:
    WinDlgBox(    HWND_DESKTOP, hwnd,
                ProdInfoDlgProc,
                NULLHANDLE, DL_PRODINFO,
                NULL) ;

    break ;
...

```

The second parameter, `hwnd`, is the handle of the client window; the call to `WinDlgBox()` takes place within its window procedure.

It is clear by reading the data delivered by SNOOPER that the owner of the dialog has become the frame window of the application rather than the client window that was specified originally.

In Chapter 4 we met the function `WinQueryWindow()`, together with its syntax and all of its flags. Among these, we had `QW_FRAMEOWNER`, which seems somewhat esoteric according to its description in the documentation. When execution reaches the code fragment dealing with the `WM_INITDLG` message in a dialog procedure, the system has already changed the owner of the dialog. This is why the value returned by `WinQueryWindow()` is different from what you might expect.

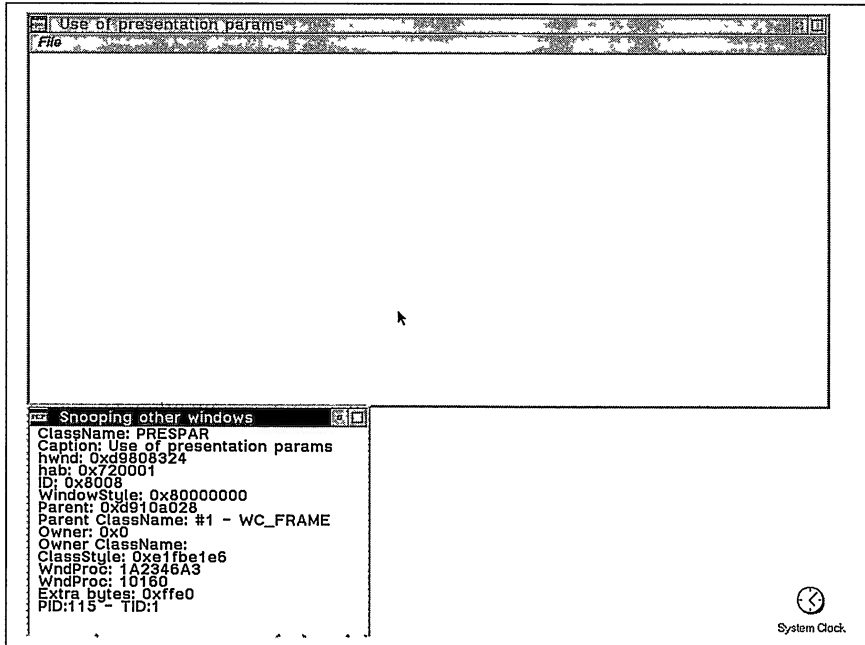


Figure 8.10 Discovering information about a window with SNOOPER.

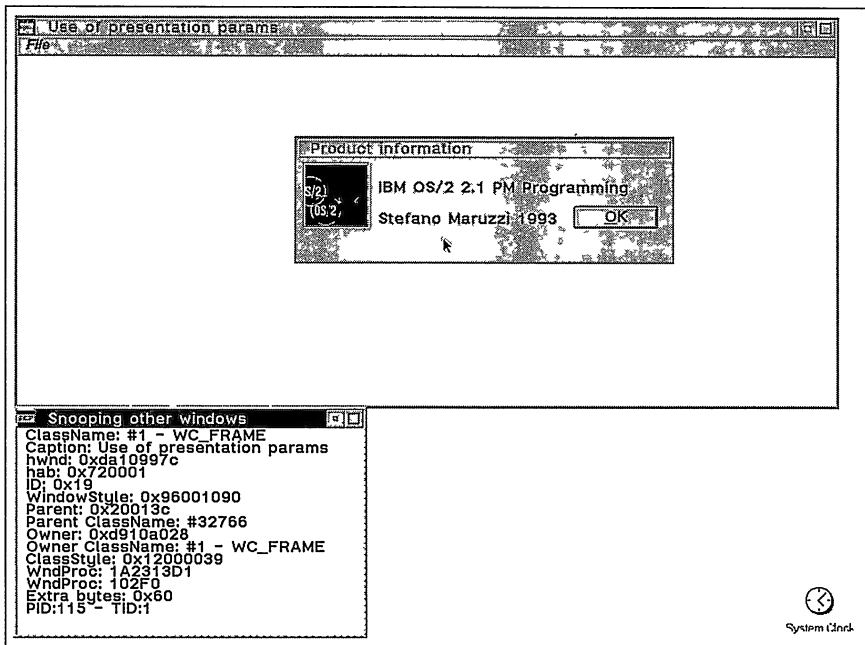


Figure 8.11 The action performed by SNOOPER on a dialog of the application illustrated in Figure 8.10.

Apart from what was specified when calling *WinDlgBox()*, the owner of a dialog is certainly the return value of the *WinQueryWindow()* function called with the *QW\_FRAMEOWNER* flag.

```
...
hwndFOwner = WinQueryWindow( hwnd, QW_FRAMEOWNER ) ;
...
```

What happens when an application calls this function directly or when PM calls it for a dialog? The first operation starts with the original owner, and then traverses the genealogy tree of that window until a sibling of *hwnd* is reached. If the parent and the owner are the same window, then the function *WinQueryWindow()* will return *NULLHANDLE*. The change of owner of a dialog is necessary if the dialog is to be modal, in which case its owner needs to be disabled. To disable a window also means to disable all its children, thus, the parent and the owner of a dialog cannot be the same window. There are several methods for getting to the handle of the owner client window of a dialog.

If you assign to a dialog the same handle for its parent and owner, even if created with *WinDlgBox()*, the dialog becomes *modeless*. This means that any actions performed outside the dialog will be accepted by the system. The dialog will lose focus and the whole logic of the application is ruined.

## Getting to Know the Owner

Getting to know the owner of a dialog is often essential for a program to perform correctly. But how do you get this information? There are two viable solutions, which are not completely interchangeable. The simpler one consists of asking for the dialog's owner and through the returned handle get to its child window that features the ID of *FID\_CLIENT*.

```
...
case WM_INITDLG:
    hwndOwner = WinQueryWindow( hwnd, QW_OWNER ) ;
    hwndParent = WinWindowFromID( hwndOwner, FID_CLIENT ) ;
    ...
    break ;
...
```

where *hwnd* is the handle of the dialog. This logic should always work, but it is not based on any granted principle and should be tested directly, application by application. Only experience suggests that the new owner of a modal dialog will be the frame window of the owner actually indicated when calling *WinDlgBox()*.

The second solution is much broader, and takes advantage of the memory area shared between the owner's window procedure and the dialog procedure. Before calling *WinDlgBox()*, you allocate a block of memory within which you can place, among other things, the handle of the application's client window. When execution flow reaches the *WM\_INITDLG* message, in the *dialog procedure* you can retrieve the information stored in this memory area by means of the value in *mp2*.

Even more simply, since in PM a handle is a plain ULONG value, you can use this address as a reference to a hypothetical shared memory area between the window procedure and the dialog procedure. Here's an example:

```
...
case MN_PRODINFO:
    WinDlgBox( HWND_DESKTOP, hwnd,
               ProdInfoDlgProc,
               NULLHANDLE, DL_PRODINFO,
               (PVOID)&hwnd );
    break ;
...

```

Inside the dialog procedure, in the code fragment dealing with the WM\_INITDLG message, you can retrieve the handle of the client window directly from mp2:

```
...
case WM_INITDLG:
    hwndClient = *((PHWND)PVOIDFROMMP( mp2)) ;
    ...
    break ;
...

```

This simple solution will *always* work, does not create any kind of problem, and requires no addition of any further code.

## Accessing Controls

Filling in the controls present in a dialog is an operation that must be performed *before* the window is displayed: The message WM\_INITDLG signals the best time for doing so. The programmer will have the dialog's handle, which is always the first parameter passed by the system to a dialog procedure, and can access to any of its controls through their ID or by getting their handle.

As we have seen in Chapter 7, inserting information in a window belonging to any of the predefined classes is an operation governed by specific messages. Sending a message in PM is essentially based on the function *WinSendMessage()*, provided you know the handle of the destination window. This piece of information is not available for controls, but can easily be retrieved through *WinWindowFromID()*.

Let's assume that there exists a listbox for displaying the names of files contained in the current directory, and that it has been described through the ID DL\_DIR. The code becomes:

```
...
hwndDir = WinWindowFromID( hwnd, DL_DIR ) ;
...

```

The handle is available only during the whole of the WM\_INITDLG message's processing. To extend its visibility, it is necessary to declare it with *static* storage class. If you are sure that you will not need this piece of information later during the dialogs management, then it is best to declare its identifier within the code block dealing with



the WM\_INITDLG message: the application will be smaller and you will not consume stack space for allocating an identifier used only once.

```
...
case WM_INITDLG:
{
    HWND hwndDir ;

    ...
    hwndDir = WinWindowFromID( hwnd, DL_DIR) ;
    ...
}
break ;
...
```

Another approach is using the *WinSendDlgItemMsg()* function:

```
#define INCL_WINDIALOGS
MRESULT APIENTRY WinSendDlgItemMsg( HWND hwndDlg,
                                     ULONG idItem,
                                     ULONG msg,
                                     MPARAM mp1,
                                     MPARAM mp2) ;
```

<i>Parameter</i>	<i>Description</i>
hwndDlg	Handle of the dialog
idItem	ID of the control
msg	Message
mp1	Parameter
mp2	Parameter
<i>Return Value</i>	<i>Description</i>
MRESULT	Return value of the dialog procedure

The syntax is absolutely identical to that of *WinSendMsg()* in its last three parameters. However, the first pair of parameters correspond to those of *WinWindowFromID()*, and that is the parent's handle (the dialog) and the ID of the control. In practice, *WinSendDlgItemMsg()* will first call *WinWindowFromID()*, and then call *WinSendMsg()* to perform its duty.

You might find it convenient to always use *WinSendMsg()* and sometimes specify directly the handle of a control previously obtained with the method just described, or even by using the CTRL( *x*, *y*) macro, which is defined in the application's header file in the following manner:

```
...
#define CTRL( x, y) WinWindowFromID( x, y)
...
```

where the two place holders are replaced, respectively, by the handle of the dialog and the ID of the control, which is the destination of the message.

To quickly get to several IDs, PM's API provides the special function *WinMultWindowFromIDs()*, which returns all the handles of a variable number of child windows, simply by specifying the handle of their owner and their IDs:

```
#define INCL_WINMESSAGEMGR
LONG APIENTRY WinMultWindowFromIDs( HWND hwndParent,
                                     PHWND phwnd,
                                     ULONG ulFirst,
                                     ULONG ulLast) ;
```

<i>Parameter</i>	<i>Description</i>
hwndParent	Handle of the owner window (i.e., the dialog)
phwnd	Pointer to a memory area within which are listed all handles of the controls within the numeric range ulFirst-ulLast
ulFirst	ID of the first control
ulLast	ID of the last control
<i>Return Value</i>	<i>Description</i>
LONG	Number of returned control handles

The second parameter of this function is a pointer to a memory area where you will insert the handles of child windows; these handles must have IDs that fall in the numeric range between ulFirst and ulLast, respectively the third and fourth parameters. The number of returned handles is equal to ulLast - ulFirst + 1. To make it efficient to use *WinMultWindowFromIDs()*, it is therefore necessary to group and assign sequential IDs to the controls belonging to the same group.

Both *WinWindowFromID()* as well as *WinMultWindowFromIDs()* can be used with any handle identifying a child or several children. PM will not distinguish in any way between a traditional window created with *WinCreateStdWindow()* or *WinCreateWindow()* and a dialog.

## The Presentation Parameters

When you create a window out of the predefined classes, you are allowed to specify a series of attributes to change its font and colors. This information is generally referred to as the *presentation parameters*, or presentation params for short. The implementation of *presparams* can take place both at the resource file level in the definition of a dialog template, as well as later during the code's execution. This second solution is generally preferred because it grants a higher degree of flexibility to the designer. The handling of the WM\_INITDLG message is the right time for changing the visual aspect of a control. Starting with OS/2 1.2 it is possible to change the *presparams* with the function *WinSetPresParam()*:

```
#define INCL_WINSYS
BOOL WinSetPresParam( HWND hwnd,
                      ULONG id,
                      ULONG cbParam,
                      PVOID pbParam) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window to which the presentation parameters refer
id	Type of presentation parameter
cbParam	Size of the area pointed at by pbParam
pbParam	Address of a memory area containing the presentation parameters' data
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure

The handle identifies the window to which the changes of the presentation parameters should be applied. The first ULONG, *id*, corresponds to the kind of *presparam* you want to change. The choices are any of the defines listed in Table 8.1.

**Table 8.1 The Values of the Presentation Parameters Available for a Control**

<i>Presparam</i>	<i>Value</i>	<i>Description</i>
PP_FOREGROUND_COLOR	1L	RGB foreground color.
PP_FOREGROUND_COLOR_INDEX	2L	Index of foreground color.
PP_BACKGROUND_COLOR	3L	RGB background color.
PP_BACKGROUND_COLOR_INDEX	4L	Index of background color.
PP_HILITE_FOREGROUND_COLOR	5L	RGB highlighted foreground color.
PP_HILITE_FOREGROUND_COLOR_INDEX	6L	Index of highlighted foreground color.
PP_HILITE_BACKGROUND_COLOR	7L	RGB highlighted background color.
PP_HILITE_BACKGROUND_COLOR_INDEX	8L	Index of highlighted background color.
PP_DISABLED_FOREGROUND_COLOR	9L	RGB disabled foreground color.
PP_DISABLED_FOREGROUND_COLOR_INDEX	10L	Index of disabled foreground color.
PP_DISABLED_BACKGROUND_COLOR	11L	RGB disabled background color.
PP_DISABLED_BACKGROUND_COLOR_INDEX	12L	Index of disabled background color.
PP_BORDER_COLOR	13L	RGB window border color.
PP_BORDER_COLOR_INDEX	14L	Index of window border color.
PP_FONT_NAME_SIZE	15L	Name and size of font.
PP_FONT_HANDLE	16L	Handle of font.
PP_RESERVED	17L	Reserved.
PP_ACTIVE_COLOR	18L	Active color.
PP_ACTIVE_COLOR_INDEX	19L	Index of active color.
PP_INACTIVE_COLOR	20L	Inactive color.
PP_INACTIVE_COLOR_INDEX	21L	Index of inactive color.

(continued)

**Table 8.1 (Continued)**

<i>Presparam</i>	<i>Value</i>	<i>Description</i>
PP_ACTIVETEXTFGNDCOLOR	22L	Foreground color for active text.
PP_ACTIVETEXTFGNDCOLORINDEX	23L	Index of foreground color for active text.
PP_ACTIVETEXTBGNDCOLOR	24L	Background color for active text.
PP_ACTIVETEXTBGNDCOLORINDEX	25L	Index of background color for active text.
PP_INACTIVETEXTFGNDCOLOR	26L	Foreground color for inactive text.
PP_INACTIVETEXTFGNDCOLORINDEX	27L	Index of foreground color for inactive text.
PP_INACTIVETEXTBGNDCOLOR	28L	Background color for inactive text.
PP_INACTIVETEXTBGNDCOLORINDEX	29L	Index of background color for inactive text.
PP_SHADOW	30L	Used to change the color when a shadow object is dropped on a control.
PP_MENUFOREGROUND_COLOR	31L	Foreground color for menus.
PP_MENUFOREGROUND_COLORINDEX	32L	Index of foreground color for menus.
PP_MENUBACKGROUND_COLOR	33L	Background color for menus.
PP_MENUBACKGROUND_COLORINDEX	34L	Index of background color for menus.
PP_MENUHILITEFGNDCOLOR	35L	Foreground color for selected menu items.
PP_MENUHILITEFGNDCOLORINDEX	36L	Index of foreground color for selected menu items.
PP_MENUHILITEBGND_COLOR	37L	Background color for selected menu items.
PP_MENUHILITEBGND_COLORINDEX	38L	Index of background color for selected menu items.
PP_MENUDISABLEDFGNDCOLOR	39L	Foreground color for disabled menu items.
PP_MENUDISABLEDFGNDCOLORINDEX	40L	Index of foreground color for disabled menu items.
PP_MENUDISABLEDBGND_COLOR	41L	Background color for disabled menu items.
PP_MENUDISABLEDBGND_COLORINDEX	42L	Index of background color for disabled menu items.
PP_USER	0x8000L	User-defined presentation parameter.

To get to know the current presentation parameters of a control, you can call the function *WinQueryPresParam()*:

```
#define INCL_WINSYS
ULONG APIENTRY WinQueryPresParam(HWND hwnd,
                                  ULONG id1,
                                  ULONG id2,
                                  PULONG pulID,
                                  ULONG cbBuf,
                                  PVOID pbBuf,
                                  ULONG fl) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window of which the presentation parameters will be changed
id1	First presentation parameter to retrieve
id2	Second presentation parameter to retrieve
pulID	Value of the indicated presentation parameter
cbBuf	Size of the pbBuf buffer
pbBuf	Buffer where the indicated presentation parameter's value will be returned
fl	Search options
<i>Return Value</i>	<i>Description</i>
ULONG	Success or failure of the operation

In addition to specifying the window's handle, you must provide the numeric ID of the type of the first and of the second parameter you wish to retrieve (*id1* and *id2*). The fourth parameter, *pulID*, will contain the ID of the presentation parameter returned between the two that are specified. The parameters are completed by a memory area pointed at by a *PVOID*, its size, and one or more flags among those listed in Table 8.2.

Finally, to remove a presentation parameter, you can call the function *WinRemovePresParam()*:

```
#define INCL_WINSYS
BOOL APIENTRY WinRemovePresParam(HWND hwnd, ULONG id) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window the presentation parameters of which will be changed
id	ID of the presentation parameter to be removed
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

where *hwnd* is the control upon which you want to act, and *id* is one of the *PP\_* defines listed in Table 8.1.

**Table 8.2** Flags of the *WinQueryPresParams()* Function

<i>Flag</i>	<i>Value</i>	<i>Description</i>
QPF_NOINHERIT	0x0001	Defines that the search for the presentation parameters must be limited only to the window identified by the handle <i>hwnd</i> .
QPF_ID1COLORINDEX	0x0002	The numeric value specified in <i>id1</i> is the index of a color: What will be returned in <i>pbBuf</i> will be the equivalent RGB color.
QPF_ID2COLORINDEX	0x0004	The numeric value specified in <i>id2</i> is the index of a color: What will be returned in <i>pbBuf</i> will be the equivalent RGB color.
QPF_PURERGBCOLOR	0x0008	Requires that the returned value be a pure RGB color.

## Setting the Presentation Parameters

Before displaying the controls present in a dialog, it is convenient to change their look by calling the function *WinSetPresParam()*. In Figure 8.12, you see a dialog with some modified controls, like the listbox background and the color of the entryfield font (although the black-and-white print will not give you an appreciation of the selected colors).



Listing 8.1 presents the source code that generates the output in Figure 8.12.

The code of *PRESPAR* is not limited to changing the look of some of the components of the dialog that is displayed after the selection of the *Dialog...* menu item off the *File* menu. Both dialogs present in the program provide additional functionality, which is not present in *PM* nor indicated in the *CUA 91* style rules, nor adopted by *WPS*. What is this all about? Nothing other than the result of some ergonomic considerations. Dialogs are often large windows that hide a great deal of the underlying client window. In general, this is not a serious problem, since the focus is on the dialog, and the user is not allowed to interact with any other part of the application. Although this is true, the user is often forced to remember some information visible on the underlying window, which might turn out to be useful even in the dialog. This is the case, for instance, of *KWIKINF.EXE*, a help utility furnished with *IBM's Toolkit*. To be certain not to write a bad name, you are often forced to move the dialog, which—fortunately—is provided with a titlebar. The addition mentioned before allows you to considerably reduce the overall region taken by a dialog, leaving visible only its titlebar (Figure 8.13) after a double-click on the right mouse button.

A subsequent double-click on the same right mouse button will restore the original dialog. You might not be *that* impressed with this new feature; but the important thing is *how* the *WM\_BUTTON2DBLCLK* message is handled within the code of the dialog procedure, demonstrating messages other than the traditional ones. The governing logic is that of storing the position of the dialog in an identifier of type *SWP* with *static*

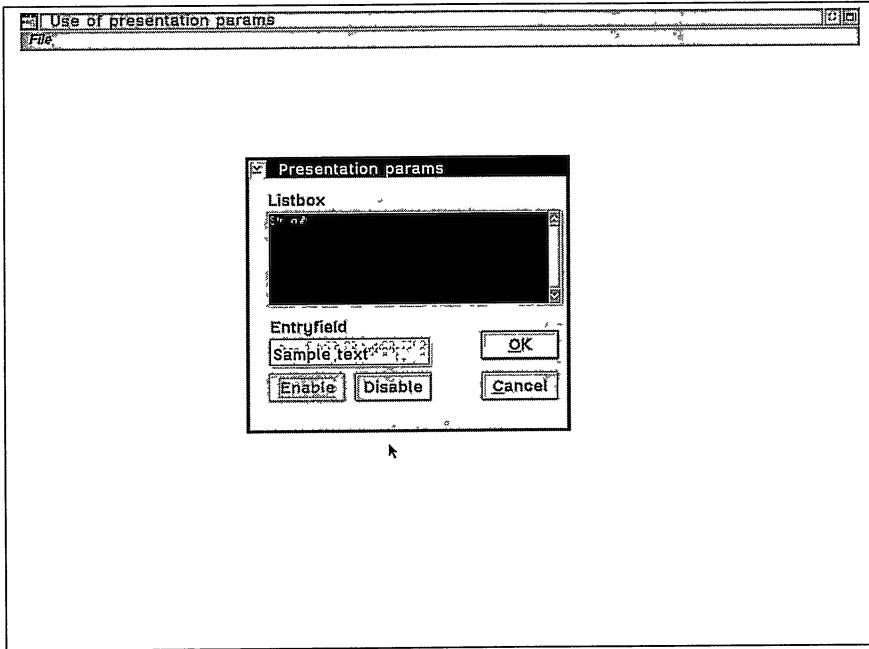


Figure 8.12 The presentation parameters of the controls present in the dialog have been changed.

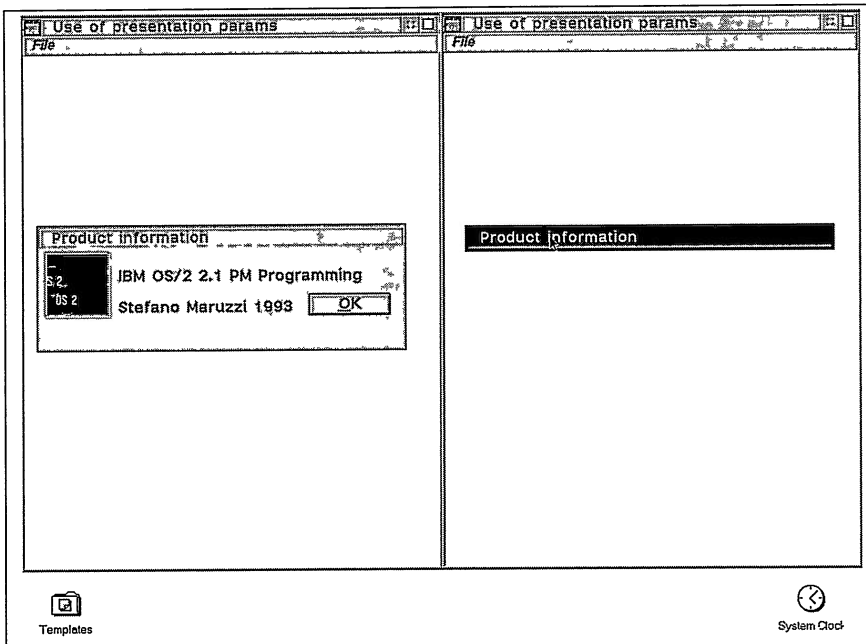


Figure 8.13 The Product information dialog after it has been reduced to the sole titlebar.

storage class, and then placing the titlebar there alone. After a second WM\_BUTTON2DBLCLK message, the dialog will be restored to its original size.

## Presentation Parameters and WinCreateWindow()

The syntax of *WinCreateWindow()* needs as its last parameter a pointer to a memory area containing the presentation parameters. This solution is very simple (at least theoretically) compared to a call to *WinSetPresParam()*, since it is executed at the same time the window is created.

The programmer only has to allocate a memory block large enough to accommodate a PRESPARAMS structure, and as many PARAM structures as there are presentation parameters to be changed.

```
typedef struct _PRESPARAMS
{ // pres
  ULONG cb ;
  PARAM aparam[ 1 ] ;
} PRESPARAMS ;

typedef PRESPARAMS *PPRESPARAMS ;
```

The PRESPARAMS structure defines in the member *cb* the overall size of the array of PARAM structures, which appears as its second member.

```
typedef struct _PARAM
{ // param
  ULONG id ;
  ULONG cb ;
  BYTE ab[ 1 ] ;
} PARAM ;

typedef PARAM *PPARAM ;
```

This structure, in turn, contains the presentation parameter of type *PP\_* in *id*, the size of the text string or of a LONG value (the *cb* member), and then the actual parameter itself (the *ab[ 1 ]* member). The memory block to be allocated must take into consideration the size of each structure, and also the size of the presentation parameter. In some cases, it will be a text string, in others it will be a four-byte object. The situation is depicted in Figure 8.14.

Summarizing, the operations to be executed are: the allocation of a memory page, the use of some pointers of the types PRESPARAMS, PARAM, and maybe even a PCHAR to compute the size of the inserted object that will have to be stored in the *cb* member of PRESPARAMS.

```
...
// allocation of a chunk of memory for the presparams
rc = DosAllocMem( (PVOID)&ppresp, 4096, PAG_COMMIT | PAG_WRITE ) ;
if( rc)
  break ;

// presentation params for the Parent & Owner pushbuttons
pparam = (PPARAM)(PVOID) ( ppresp - aparam ) ;
p = (PCHAR)pparam ;
```



```

// set the values
ppresp -> aparam -> id = PP_FONTNAMESIZE ;
strcpy( pparam -> ab, szFont) ;
pparam -> cb = sizeof( szFont) ;
lDelta = sizeof( szFont) + sizeof( ULONG) * 2 ;
ppresp -> cb += lDelta ;

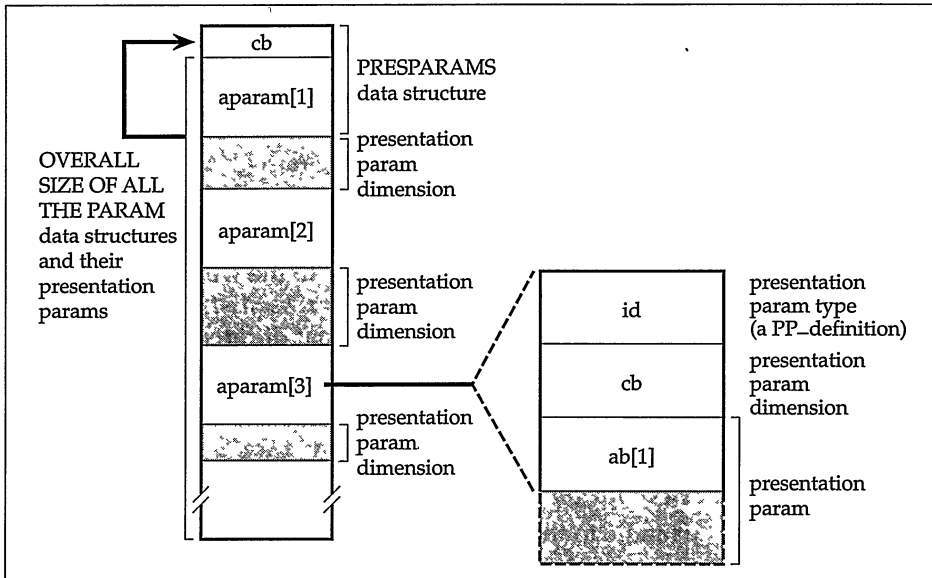
// move the pointer at the end of the first presparam
p += lDelta ;
pparam = (PPARAM)p ;

// set the values
pparam -> id = PP_FOREGROUNDCOLORINDEX ;
pparam -> cb = sizeof( COLOR) ;
*((COLOR *)pparam -> ab) = CLR_WHITE ;
lDelta = sizeof( COLOR) + sizeof( ULONG) * 2 ;
ppresp -> cb += lDelta ;

// move the pointer at the end of the second presparam
p += lDelta ;
pparam = (PPARAM)p ;

// set the values
pparam -> id = PP_BACKGROUNDCOLORINDEX ;
pparam -> cb = sizeof( COLOR) ;
*((COLOR *)pparam -> ab) = CLR_RED ;
lDelta = sizeof( COLOR) + sizeof( ULONG) * 2 ;
ppresp -> cb += lDelta ;
...

```



**Figure 8.14** Scheme for managing a block of memory containing a number of presentation parameters.

When this preparatory phase is over, you only have to pass the pointer to *WinCreateWindow()* as its last parameter:

```
...
hwndCtl = WinCreateWindow( ..., ppresp) ;
...
```

Finally, you must destroy the allocated memory block:

```
...
DosFreeMem( ppresp) ;
...
```

Chapter 13 contains a significant example of how to assign different presentation parameters by this approach.

## *Presentation Parameters and Resource Files*

The same defines—introduced by the prefix *PP\_*—that are used in the source code can be used in the resource files with the *MENU*, *DLGTEMPLATE*, *WINDOWTEMPLATE*, and other directives. The *RC.EXE* resource compiler also supports the *PRESPARAMS* directive with the following syntax:

```
PRESPARAMS PP_xxx, presentation attribute
```

In Listing 8.2, implementing a first version of the *WHEREIS* utility, you will find inside a *WINDOWTEMPLATE* the use of *PRESPARAMS* to overcome an aesthetic problem due to the background color of the entryfields in a template. The adopted color is always light gray, which is not appropriate when the rest of the window is white. Thanks to the presentation parameter *PP\_BACKGROUNDINDEX* in the window or dialog template, it is possible to assign the same color used for the remaining parts of the window (white in this case).

## *Terminating a Modal Dialog*

The function *WinDlgBox()* does not return any value to the window procedure from which it was called, at least not until the function *WinDismissDlg()* is executed in the dialog procedure. This is yet another characteristic that differentiates a dialog procedure from a window procedure.

```
#define INCL_WINDIALOGS
BOOL WinDismissDlg(HWND hwndDlg, ULONG ulResult) ;
```

<i>Parameter</i>	<i>Description</i>
<i>hwndDlg</i>	Handle to the dialog
<i>ulResult</i>	Return value of the dialog procedure

<i>Return Value</i>	<i>Description</i>
<i>BOOL</i>	Success or failure

It is not possible to establish in absolute terms the physical position of *WinDismissDlg()* in a dialog procedure, although it will generally be found in the code dealing with the *WM\_COMMAND* message. More precisely, the call to *WinDismissDlg()* concerns the interception of the conditions of the OK and Cancel buttons of the dialog. It is the designer's responsibility to define the termination conditions of the dialog, from when the template is written, until adequate code is implemented.

```

...
case WM_COMMAND:
    switch( SHORTIFROMMP( mp1))
    {
        case DID_OK:
            WinDismissDlg( hDlg, TRUE ) ;
            return (MRESULT)1L ;

        case DID_CANCEL:
            WinDismissDlg( hDlg, FALSE ) ;
            return (MRESULT)0L ;

    }
    break ;
...

```

For a *modal* dialog loaded with *WinDlgBox()* it is not possible to use *WinDestroyWindow()*, an operation that is permitted with *modeless* dialogs. The defines *DID\_OK* and *DID\_CANCEL* are present in *PMWIN.H*, and are usually used to indicate the termination conditions of a dialog for positive actions (*DID\_OK*) or for cancellation (*DID\_CANCEL*). The values in *PMWIN.H* are the following:

```

#define DID_OK      1
#define DID_CANCEL  2

```

When terminating a dialog with *WinDismissDlg()* you must indicate in the second parameter the value *TRUE* when dealing with a *DID\_OK* condition, or *FALSE* for *DID\_CANCEL*. The value specified as the second parameter of *WinDismissDlg()* will be returned by *WinDlgBox()* in the window procedure.

## Default Message Processing

Each message passing through a dialog must be subject to some default processing. The function *WinDefDlgProc()* will take care of this task.

```

...
switch( msg)
{
    ...
}
return WinDefDlgProc( hwnd, msg, mp1, mp2 ) ;
}

```

As for window procedures, it is preferable to place *WinDefDlgProc()* outside the `switch` block processing messages, so as to make sure that all messages will get to the default processing provided by the system.

## Some Considerations

The activation of a modal dialog with *WinDlgBox()* will cause input focus to be captured from the owner window. This will depend on the nature of the *WinDlgBox()* function: when it is executing it will prevent any messages that are not addressed to itself from reaching the application's message queue, and thus grant itself exclusive dominance over the user's input. Later, when the *WinDismissDlg()* is called, the window is not yet destroyed. *WinDlgBox()* will always perform this operation before returning a value to the calling window procedure.

Furthermore, as you can see in Listing 8.1, the function *WinDefDlgProc()* will exhibit rather unusual behavior concerning the default processing of the `WM_COMMAND` message. *WinDefDlgProc()* will always destroy the dialog, whatever the source of the message. Given that this is the ideal behavior when the OK or the Cancel button is pressed, it is not an optimal way for handling any other possible pushbuttons present inside the dialog. You have to take care of this when implementing the dialog procedure.

Never forget that you must always export the name of a dialog procedure to the application's module definition file. A dialog procedure, just like a window procedure, is a function of the `EXPENTRY` type, so it must always be present in the `EXPORTS` section of the `DEF` file.

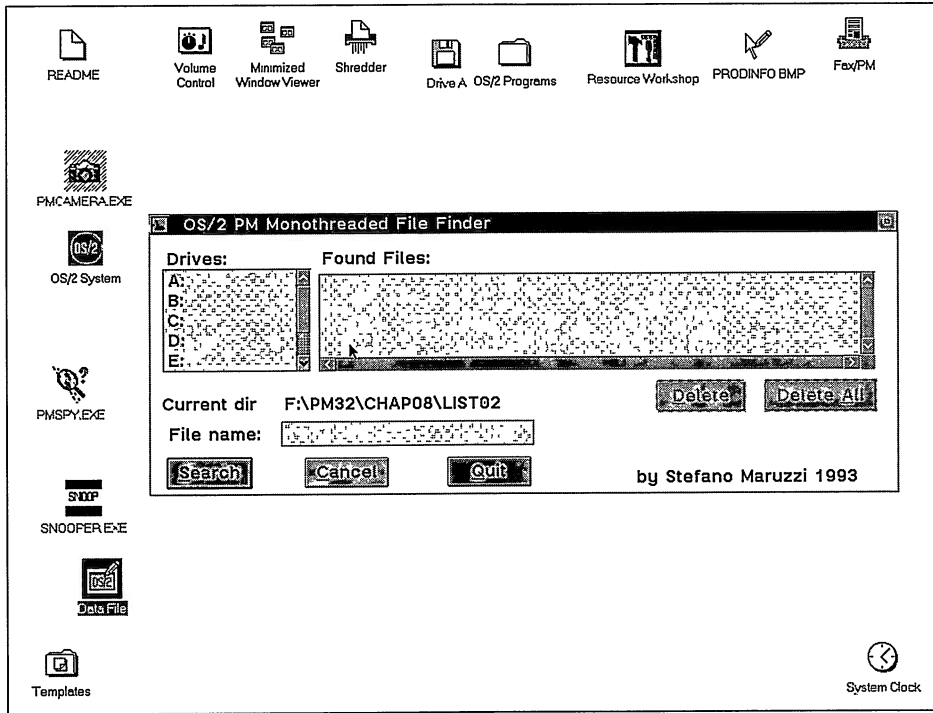
---

## A Utility for PM

Now we know about all the components that will let us build a PM application that is more significant than the previous tutorial examples. A problem present since the very first versions of PM is that of the generation of support files during the compilation and linking phases of development. This is even more evident in those cases when you want to take advantage of the advanced features of Borland's compiler, like the generation of precompiled headers. The directory containing the examples will be overcrowded with files that are vital during the development phase, but that later just take up disk space. You need to be able to get rid of all these with one command and reclaim the unused disk space. The `WHEREIS` application shown in Figure 8.15 will accomplish this.

The application of `WHEREIS` cannot be resized manually in order to stop the user from hiding from view some essential controls for the program's correct operation. However, a minimize icon is present so that you can make the application disappear temporarily.

In the upper part of the window there are two listboxes for displaying the drives that are present in the system and the files that have been found after the user's specifying some search criteria.



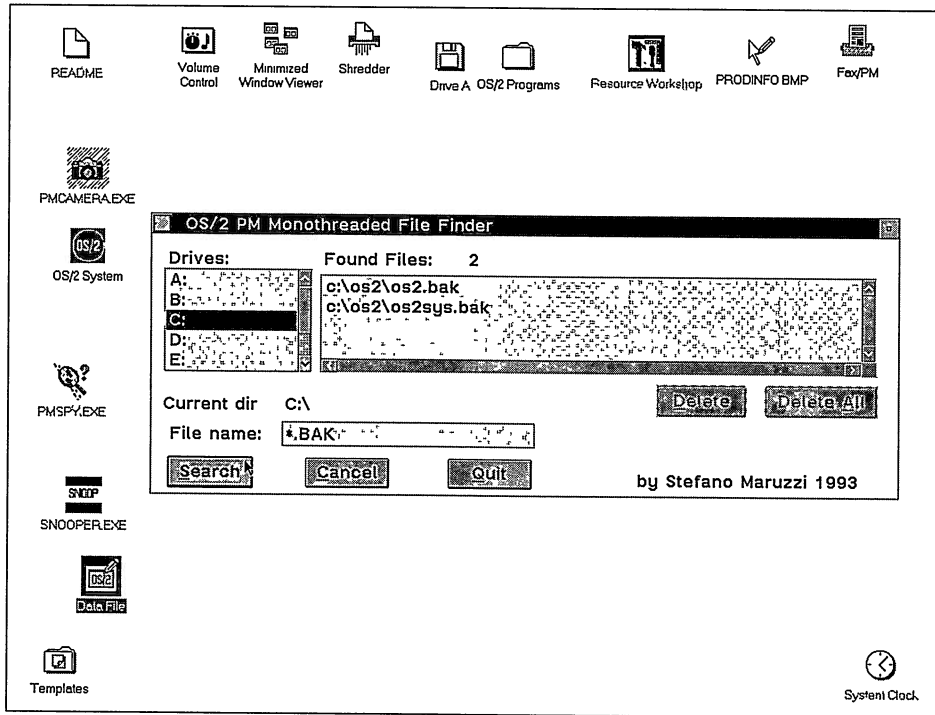
**Figure 8.15** The window of the WHEREIS application contains several controls specialized for searching a file in a drive of the system.

Beneath the listboxes you will find the complete *pathname* of the current directory, which will show you where you are in the file system. Nonetheless, the search for a file will be conducted on the whole drive, rather than on a directory alone. To the left of this, there will appear two pushbuttons with the labels *Delete* and *Delete All*; these two pushbuttons are initially disabled. They will be enabled automatically by the application as soon as the search operation finds some positive matches.

The search for a file or for a set of files (the usual wildcard characters are fully supported) requires the user to write the search criteria in the entryfield that holds input focus when the application is started. In the lower part of the window there are three buttons for starting a search, canceling the search request, and quitting the application.

## Searching for a File

To search for a file, the user will have to input its name in the entryfield, as shown in Figure 8.16, then select the *Search* button. The application is also provided with a series of accelerators (*ALT+S* for *Search*, *ALT+C* for *Cancel*, and *ALT+Q* for *quit*), to facilitate keyboard operations.



**Figure 8.16** Search for files with the BAK extension in the C drive of the system.

Don't get confused by the way the window is created. This is *not* a dialog, and its behavior is different. Notice the button with the style BS\_DEFAULT. This attribute is assigned to Search, however, pressing the Enter key after typing some text in the entryfield will not cause this button to be selected automatically. To overcome this, the code provides a special accelerator that is equivalent to pressing the Enter key. Any time the Enter key is pressed, the Search button will be selected. Be careful, though! In the 102-key keyboards there are two Enter keys: the traditional one and the one on the numeric key pad. The two keys, although being theoretically equivalent, are handled by PM with two distinct defines: VK\_NEWLINE and VK\_ENTER. In the resource file of WHEREIS you will therefore find that these two accelerators will exhibit the same behavior typical of a dialog, even in this ordinary window.

During the search, the cursor changes into the hourglass icon to stress even visually that the application is busy in its processing phase. Once a file has been found, it will be displayed directly in the appropriate listbox, and the count of matches will be incremented.

This listbox is also provided with the style LS\_EXTENDEDESEL to allow the user to select more than one file at the same time. When the user clicks the left mouse button over a file, he will indirectly enable the Delete and Delete All buttons. These two

pushbuttons are associated with accelerators, Ctrl+D and Ctrl+A, respectively. There are two basic actions that the user can perform: execute a selected file by selecting it and confirming the choice (double mouse-click), or delete one or more files. Executing a file is possible only if it represents an executable module with the EXE extension, while there are no special restrictions for deleting a file.

## *The Scheme of the Application*

Several controls are positioned in the client area of the application. To make operations simpler, a window template has been written in the resource file; it is later loaded by means of a call to the function *WinLoadDlg()*, already described in the previous listings. The main window of the application belongs to the class *WHEREIS*, a class registered by the application. This means that the resource file must also define this name in the window template, and, above all, it means that you cannot rely upon receiving the message *WM\_INITDLG* for customizing the look of the controls contained in the window. It is the application itself that takes care of this aspect, by implementing a different solution. The mechanism is very simple, and consists of sending a message defined in the code to the window procedure in order to prepare all controls before the window is actually displayed. In the window template there is no *FCF\_SHELLPOSITION* flag in order to prevent the execution of *WinLoadDlg()* from displaying the window. For this purpose, the application relies on the services of *WinSetWindowPos()*, which is called immediately after sending the *WM\_COMMAND* message with the special value of *MN\_STARTUP* defined in the code.

```
...
WinSendMsg( CLIENT( hwndFrame), WM_COMMAND,
            MPFROMSHORT( MN_STARTUP), 0L );
...
```

*MN\_STARTUP* is defined in the application's header file, just like an ID assigned to a menu item. There is a corresponding case branch that will be taken only when that message is sent by *main()*; the purpose of this portion of code is principally that of capturing in identifiers with static storage class the handles of all controls.

```
...
case MN_STARTUP:
{
    ULONG u1DriveNum, i = 0 ;
    ULONG u1Drives ;
    CHAR szBuffer[ 10] ;

    // retrieve control handles
    hwndDrive = WinWindowFromID( hwnd, ID_DRIVE) ;
    hwndDir = WinWindowFromID( hwnd, ID_DIR) ;
    ...
    hwndFound = WinWindowFromID( hwnd, ID_FOUND) ;
    DosQueryCurrentDisk( &u1DriveNum, &u1Drives) ;
}
```

```

while( i < 26 && u1Drives)
{
    if( u1Drives & 1)
    {
        sprintf( szBuffer, "%c:", 65 + i ) ;
        WinSendMessage( hwndDrive, LM_INSERTITEM,
            MPFROMSHORT( LIT_SORTASCENDING),
            MPFROMP( (PSZ)szBuffer)) ;
    }
    u1Drives >= 1 ;
    i+ + ;
}
sprintf( szBuffer, "%c:", u1DriveNum + 64) ;
WinSetWindowText( hwndDrive, (PSZ)szBuffer) ;
ShowCurDir( u1DriveNum - 1, hwndCurDir) ;
WinSetFocus( HWND_DESKTOP, hwndEdit) ;
}
break ;
...

```

All the identifiers of type `HWND` are declared with `static` storage class, so that they are always accessible during the handling of a message in the window procedure. This situation would have been ideal even for `WinMultWindowFromIds()`, provided that all IDs were sequential. However, the one-at-a-time approach used with `WinWindowFromID()` was preferred because it allows you to deal with any handle in separate identifiers, and is easier to understand and remember than in a single array of `hwnd` that would be accessed only through cryptic indexes. (Changing the code in this sense would be an excellent exercise, though. Try it!).

After retrieving the handles of all controls, the program will detect all drives present in the system by calling the function `DosQueryCurrentDisk()` that returns in the second `ULONG` a map of the devices installed in the computer.

```

#define INCL_DOSFILEMGR
APIRET APIENTRY DosQueryCurrentDisk( PULONG pDriveNumb,
                                     PULONG pLogicalDriveMap) ;

```

<i>Parameter</i>	<i>Description</i>
<code>pDriveNumb</code>	Number of drives present in the system
<code>pLogicalDriveMap</code>	Map of the drives present in the system
<i>Return Value</i>	<i>Description</i>
<code>APIRET</code>	Success or failure of the operation

The `while` loop allows you to find the letters that identify each drive and display them in the appropriate listbox. The function `ShowCurDir()`, not documented in the previous code fragment, will simply select in the listbox the current drive and display the current directory. The preparatory phase ends with assigning focus to the entry-field so that the user can type in the name of the file to be searched for.





The first operation that is performed when the notification code `LN_ENTER` is received is identifying what item was selected in the listbox. On the basis of the return value of `WinSendMsg()`, the program retrieves the selected text string (the file name) and its length:

```
...
sItem = (short)WinSendMsg( hwndDir, LM_QUERYITEMTEXT,
                          MPFROM2SHORT( sPos, sizeof( szBuffer)),
                          MPFROMP( szBuffer)) ;
...
```

Once the string is available, it is necessary to check if its extension is EXE, and in that case execute the corresponding file. The `strcmpi()` library function will return a value of zero when two strings are equal. In case of failure this portion of code is abandoned without calling `DosExecPgm()`:

```
...
// skip if not an EXE
if( strcmpi( ( szBuffer + ( sItem - 3)), "EXE"))
    break ;
DosExecPgm( szString, sizeof( szString),
            EXEC_ASYNC,
            NULL, NULL,
            &rc,
            szBuffer) ;
...
```

The execution of the new application is obviously asynchronous in relation to its parent, so that it will have a completely autonomous life with respect to the parent process.

## Selection of a File

Once a search has been executed, the names of all matching files appear in the appropriate listbox. When periodically cleaning up the hard disk, you will probably be interested in deleting all BAK, OBJ, RES, and other files that consume space. The listbox is provided with the flag `LS_EXTENDEDSEL`, and will therefore allow the user to select more than one file at a time. The notification code `LN_SELECT`, originating from the user's interactions with this listbox, is issued by the application to change the status of the Delete and Delete All buttons. These buttons are enabled as soon as one item is selected, and disabled when no items are selected.

```
...
if( ( sItem = (short)WinSendMsg( hwndDir, LM_QUERYSELECTION,
                                MPFROMSHORT( LIT_FIRST),
                                MPFROMLONG( 0L))) != LIT_NONE)
{
    WinEnableWindow( hwndDel, TRUE) ;
    WinEnableWindow( hwndDelAll, TRUE) ;
}
```

```

}
else
{
    WinEnableWindow( hwndDel, FALSE) ;
    WinEnableWindow( hwndDelAll, FALSE) ;
}
...

```

The buttons are enabled several times in case of multiple selections. In each case, when the notification code `LN_SELECT` is received, there is no simple and immediate way of knowing if it refers to some item different from the previous one, since this is a listbox with `LS_EXTENDEDSEL`. Rather than implementing complex algorithms for determining this, we take a simple and straightforward route, even if its actions are somewhat redundant.

## Deleting Files

This portion of the program is both interesting and dangerous. Never ever use the `WHEREIS` utility with important files like `C` or `RC`; if you press the wrong keys and have no updated backups available, then you're ruined!

The selection of an item in the listbox displaying the result of the search, will automatically enable the Delete and Delete All buttons. If you press the first one, you will delete *all selected files*, with Delete All you will delete *all files listed in the listbox*; and here, the word "delete" refers to the magnetic media, not to the contents of the listbox! So, be careful.

The code that takes care of these two ways of handling deletions is practically identical, although the action is more selective in the first case. The only distinction concerns the selection of the whole listbox when Delete All is pressed.

```

...
sItem = (short)WinSendMsg( hwndDir, LM_QUERYITEMCOUNT, 0L, 0L) ;
for( sPos = 0; sPos < sItem; sPos++ )
    WinSendMsg( hwndDir, LM_SELECTITEM,
                MPFROMSHORT( sPos),
                MPFROMSHORT( TRUE) ) ;
...

```

The delete all case does not have a break delimiting its end, so its processing will proceed with whatever code follows, and that is the code handling the Delete button:

```

...
sStart = LIT_FIRST ;
WinSendMsg( hwndDir, LM_SETTOPINDEX,
            MPFROMSHORT( 0), 0L) ;
...

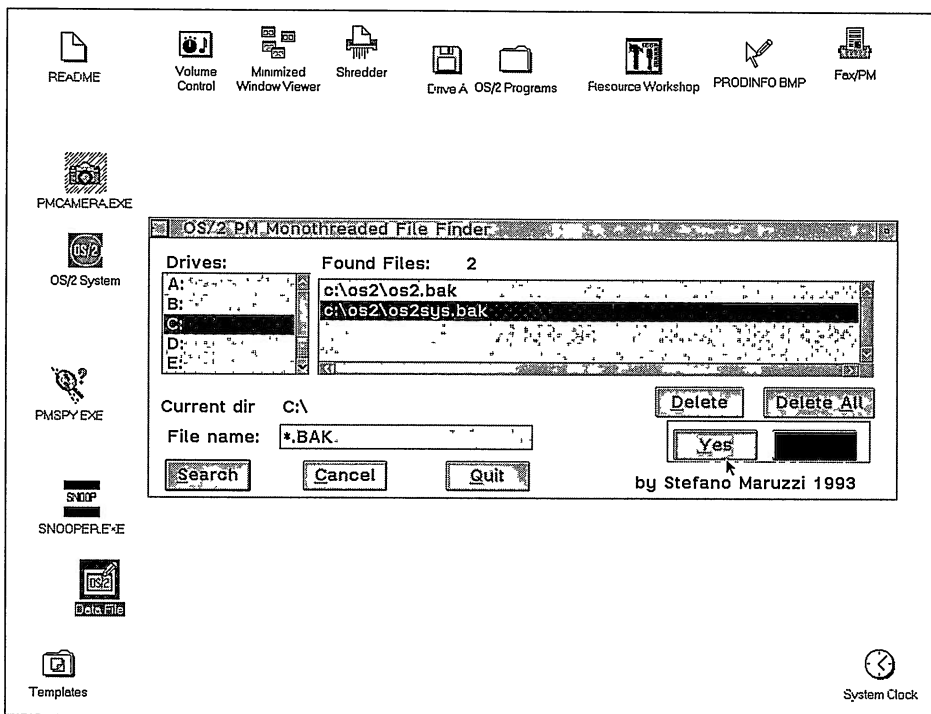
```

The first operation performed by the code is to make visible as the listbox's top-index item, the first one physically available (item 0). The user could have performed a vertical scroll of the window's contents when selecting a file. Although

this operation does not change the order of the items in the listbox, it does, however, induce erroneous behavior in the selection of the appropriate listbox item, probably due to some internal bug. To overcome this odd problem, we resort to a solution that is just as odd, because it is not logically consistent: making the first item the topmost. At this point everything is ready for deletion. A question to the user is mandatory before proceeding with this risky operation: Are you really sure you want to continue? Since often the answer will be negative, we decide to suspend the program's execution indefinitely by displaying a modal dialog with a rather unusual look (Figure 8.17).

The dialog, by capturing the input focus, will impede the execution of any other operation in the application, so it does indeed achieve the intended purpose. The code for displaying the dialog and getting its return value is very simple and terse:

```
...
if( !WinDlgBox(  HWND_DESKTOP, hwnd,
                YesNoProc,
                NULLHANDLE, DL_CONFIRM,
                NULL))
    break ;
...
```



**Figure 8.17** Before proceeding with the deletion of the selected files, the user must press the Yes button that appears in the window.

If the return value is FALSE, then execution of the code of the two buttons Delete and Delete All is skipped, thus canceling the requested deletion. The Yes button has a green background, while the No button has a red background. It is in the *YesNoProc()* dialog procedure that the color attributes are assigned when the message WM\_INITDLG is received:

```

...
case WM_INITDLG:
{
    COLOR clr = CLR_GREEN ;

    WinSetPresParam( CTRL( hwnd, DID_OK), PP_BACKGROUNDCOLORINDEX,
                    sizeof clr, &clr) ;

    clr = CLR_RED ;
    WinSetPresParam( CTRL( hwnd, DID_CANCEL), PP_BACKGROUNDCOLORINDEX,
                    sizeof clr, &clr) ;
}
break ;
...

```

The CTRL macro returns the handle of each pushbutton, starting from the handle of the dialog and the ID assigned to it in the resource file. The confirmation of the delete operation is allowed only once, before initiating the operation. To make the utility even safer, it is advisable to transfer the confirmation request directly into the deletion algorithm, and thereby ask for confirmation for each file about to be deleted. (You might try to implement this change as an exercise!)

The delete operations are placed in a while loop based on the value returned by the message LM\_QUERYITEMSELECTION to the listbox containing the files. The listbox will return the index of the selected item, or LIT\_NONE if there is no selection. The search for the first highlighted displayed item must always start from the origin of the listbox, even though this is an LS\_EXTENDESEL listbox. This behavioral variation is indicated by the application's design. In general, in an *extended selection*, only the first selected item can be identified, starting with LIT\_FIRST. For all other items, the search must progress from the index of the previous one. This is not the case here, because as soon as the index of the first selected item is retrieved, that item is deleted from the listbox together with the corresponding file in the file system. Hence, the item that originally appeared as the *second* selected item, now becomes the *first* one, after the deletion of what was originally the first selected item.

```

...
while( ( sPos = (short)WinSendMsg( hwndDir, LM_QUERYSELECTION,
                                MPFROMSHORT( LIT_FIRST),
                                0L)) != LIT_NONE)
{
    WinSendMsg( hwndDir, LM_QUERYITEMTEXT,
                MPFROM2SHORT( sPos, sizeof( szBuffer)),
                MPFROMP( szBuffer)) ;
}

```

```

    DosQueryPathInfo(  szBuffer, FIL_STANDARD, &fs3, sizeof( fs3)) ;
    if( fs3.attrFile & ( FILE_NORMAL | FILE_ARCHIVED))
    {
        rc = DosDelete( szBuffer) ;
        sItem = (short)WinSendMessage( hwndDir, LM_DELETEITEM,
                                     MPFROMSHORT( sPos), 0L) ;
        sprintf( szBuffer, "%d", sItem) ;
        WinSetWindowText( hwndFound, szBuffer) ;
    }
}
...

```

The removal of an item from the listbox is done by sending the message LM\_DELETEITEM. The return value of *WinSendMessage()* corresponds to the number of remaining items in the window. The application will immediately update the number of files left in the listbox before proceeding with a new iteration in the deletion loop. This piece of information is then used to disable the delete buttons and to clear the search criteria previously inserted into the entry field (since there will no longer be any file of that type in the selected drive).

```

...
// check if empty
if( !sItem)
{
    WinEnableWindow( hwndDel, FALSE) ;
    WinEnableWindow( hwndDelAll, FALSE) ;
    WinSetWindowText( hwndEdit, "" ) ;
    WinSetFocus( HWND_DESKTOP, hwndEdit) ;
}
...

```

If the request for confirming the deletion were executed for each single file in the listbox, it would have been necessary to introduce a minor modification in the code. In fact, in order not to change the working logic of the *while* loop, it would have been necessary that the not yet deleted selected items lose their highlighting, in order to prevent them from being presented for confirmation over and over again through each subsequent run through the loop, and so freezing the application.

The entire source code of WHEREIS appears in Listing 8.2.



## Searching for Files

A very important task in the WHEREIS application is that performed by the file system search algorithm. The code of this portion of the program is external to the PM module of WHEREIS, but it is not a separate thread from the application. The entire logic of the search algorithm is based on the functions *DosFindFirst()* and *DosFindNext()*. In order to operate correctly in an OS/2 system that employs the HPFS file system, it is necessary to introduce some changes to the code to be able to access those files that have extended attributes and long names. But this an exercise left to the reader!

## Creating an Open Box

A task that shows up in almost every OS/2 application is that of letting the user select a file to be loaded into memory. Generally, for this purpose an *Open extended command* option will be available in the File top-level menu.

Building an Open box—this is the jargon for a modal dialog that allows you to select and load a file—is a simple operation. Although it is impossible to standardize its look, such a dialog will always have a listbox, some static windows, and one or more entryfields. The listbox will contain the list of files, the directory path, and the system drive from which the selection is being made. A recent trend is toward a more complicated and sophisticated look as well as greater functionality, which better caters to the growing number of drives present in personal computers, and better helps the user in making selections. The first consideration is how to create a dialog that looks good and works well. In Figure 8.18 you can see the dialog that is implemented in the application of Listing 8.3.



The contents of the current directory are listed in the listbox labeled with the text Files. This control is large enough to accommodate file names conforming to the rules of FAT (8 characters plus 3 for the extension), and it should probably be made larger if you were to use long file names, like those recognized by the HPFS. The listbox on

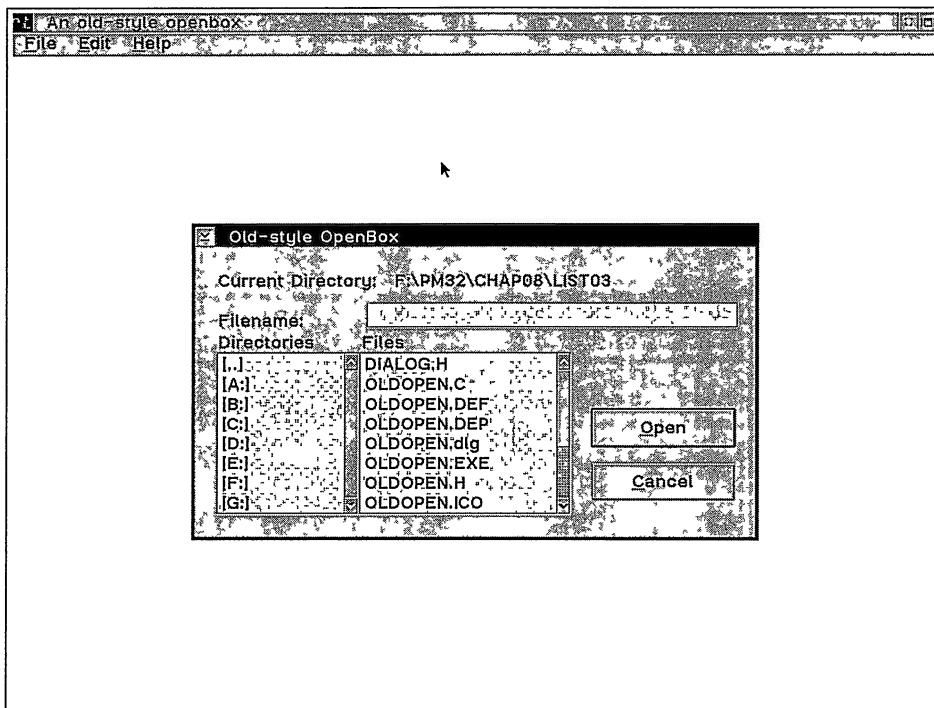


Figure 8.18 A typical modal dialog used to select and load a file from disk.

the lower left side of the window lists the names of the drives present in the system and the subdirectories of the current one (if any).

The name of the current directory is shown in a static control in the upper part of the dialog. Under this window there appears an entry field that is ready to accept the user's keyboard input. The dialog is then completed by two pushbuttons that either confirm the selection or cancel the operation.

The separation of file names from other information makes it easier for the user to perform selections, as it prevents having to keep up with ongoing vertical scrolling, which is the rule in open boxes that have only one listbox.

## *Positioning the Dialog*

The generic position of a dialog is defined in its template, and is expressed in coordinates of the parent window. As this will be the screen, the position of the dialog is often independent of the position of the application window, especially when this one is not maximized. In this case, though, we implement an algorithm to position the dialog at the center of the application's client window. There is no way to be sure that this operation will succeed, because it depends on the actual size of the main window on the screen. The algorithm computes an experimental position and then checks it against the size of the application's client window. The computation is based on the measurement of both the dialog's size and the client's. The first is easy to obtain, since the handle of the window is available when the message WM\_INITDLG is handled in the dialog procedure. To get to the handle of the client window, it is necessary to declare a source file scope identifier (which we will *never* do in this book), or resort to one of the information-passing techniques previously described (which is the right way of doing things!).

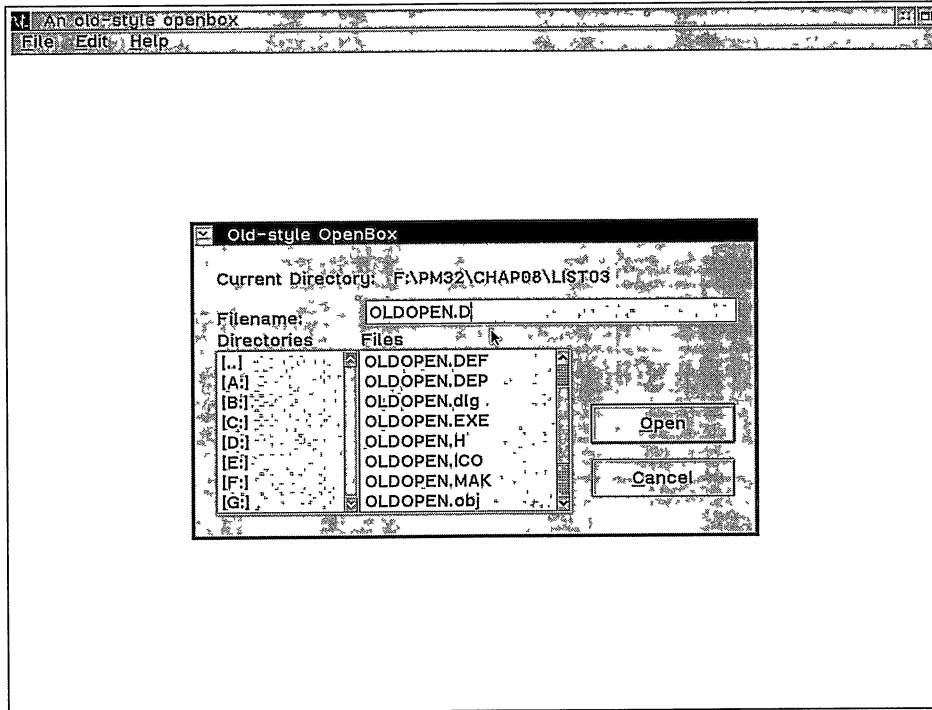
## *A New Data Type*

In the dialog procedure it is necessary to know the handle of the client window; then we need to let the client window have the name of the file by returning its complete pathname. This is a typical situation that requires a shared memory area to be defined and accessible to both procedures for exchanging information in both directions (Figure 8.19). To handle data in the shared memory area we define a new data type in the application's header file:

```
...
typedef struct _DATA
{
    CHAR szFileName[ 80] ;
    HWND hwndOwner ;
} DATA ;

typedef DATA * PDATA ;
...
```





**Figure 8.19** The application will force the listbox to scroll its contents accordingly to the text typed into the entryfield.

The array of CHAR will hold the complete name of the selected file, while the handle will refer to the application's client window (here it has been called `hwndOwner`, since the client should in theory play the role of the dialog's owner). In the window procedure there appears the declaration of an identifier of type DATA with static storage class. The message handling code for `WM_CREATE` will assign to the `hwndOwner` member the handle of the client window, which will eventually be used in other portions of the code.

## Centering the Dialog

In order to position the dialog at the center of the client, a simple algorithm will compute its position on the basis of the sizes of the two rectangles obtained through `WinQueryWindowRect()`. Before calling `WinSetWindowPos()` to position the open box, you must convert the position into screen coordinates, which is the right unit, as the desktop is the dialog's parent.

```
...
WinQueryWindowRect( hwndOwner, &rc1) ;
WinQueryWindowRect( hwnd, &rc2) ;
```

```

ptl.x = (( rc1.xRight - rc1.xLeft) - ( rc2.xRight - rc2.xLeft)) / 2 ;
ptl.y = (( rc1.yTop - rc1.yBottom) - ( rc2.yTop - rc2.yBottom)) / 2 ;

// centered dialog
if( ptl.x > 0 && ptl.y > 0)
{
    WinMapWindowPoints( hwndOwner, HWND_DESKTOP, &ptl, 1) ;
    WinSetWindowPos( hwnd, HWND_TOP,
                    ptl.x, ptl.y,
                    0, 0,
                    SWP_MOVE) ;
}
...

```

## Filling in Controls

The PM API, although rich in functions and messages for dealing with the most disparate requirements, does not have a specific tool for transferring the contents of a directory directly into an open box. The whole chore is on the programmer's shoulders, and this will allow a degree of customization that would be difficult to achieve through API functions only.

So, a specific function has been written for filling in the listbox, *ListFilesDirs()*, which is essentially based on the usage of the *Dos* functions that we have already met in the description of WHEREIS. Resorting to an external function responds to a precise design criterion of the program. An open box is an area of continuous interaction between the user and the application, and therefore subject to frequent and rapid changes pertaining to the contents of the various controls it contains. It is reasonable to expect that the operation of filling in the listbox will happen more than once during the time the dialog is on the screen, rather than just when the window is initialized.

## Input Sources

Writing an open box is an excellent exercise for the programmer because of the high degree of interaction that takes place among the application, the user, and the various controls. The changes to the dialog's output can originate from an input directly in the entryfield, or from selections of items in the listbox, and the selection of the drive, the directory, and the files. For each of the two listboxes, both kinds of selection actions are supported: simple selection and selection with confirmation (double-click). The selection of the name of a directory or a drive in the corresponding listbox does not generate any kind of apparent effect on the program. The same action in the Files listbox, on the other hand, will cause the selected item to be displayed in the entryfield.

A double-click on the name of a drive or a directory will cause the current directory to be abandoned, and the selected one to be displayed. A double-click on a file name implies its selection and the notification to the window procedure, with the consequential loading operation (which, though, is not implemented in Listing 8.3). For a listbox, a double-click of the left mouse button is equivalent to pressing the Enter key.

The text typed by the user in the entryfield can take on different forms—in the dialog procedure there must be some code knowledgeable enough to interpret its meaning. Assuming that the user does not commit any input error, the following scenarios are possible:

- Direct input of a file name
- Input of a drive name
- Input of a directory name
- Input of a drive name and a directory name

The entryfield will communicate with the dialog procedure by sending it notification codes through the `WM_CONTROL` message. The same is true for the two listboxes, while the two pushbuttons, OK and Cancel, will use `WM_COMMAND`. When the user types in some text in the entryfield, a piece of code dealing with the notification code `EN_CHANGE` will be called automatically to perform the scrolling of the contents of the listbox of files, provided there is a file starting with the same characters.

In Figure 8.18, the list of files contains a number of names starting with the letter O, and only one starting with the letter D. If the user types the letter O (case doesn't matter), then the entryfield will force the listbox to scroll its contents so as to display as its first element the item with the lowest index starting with that letter (Figure 8.19).

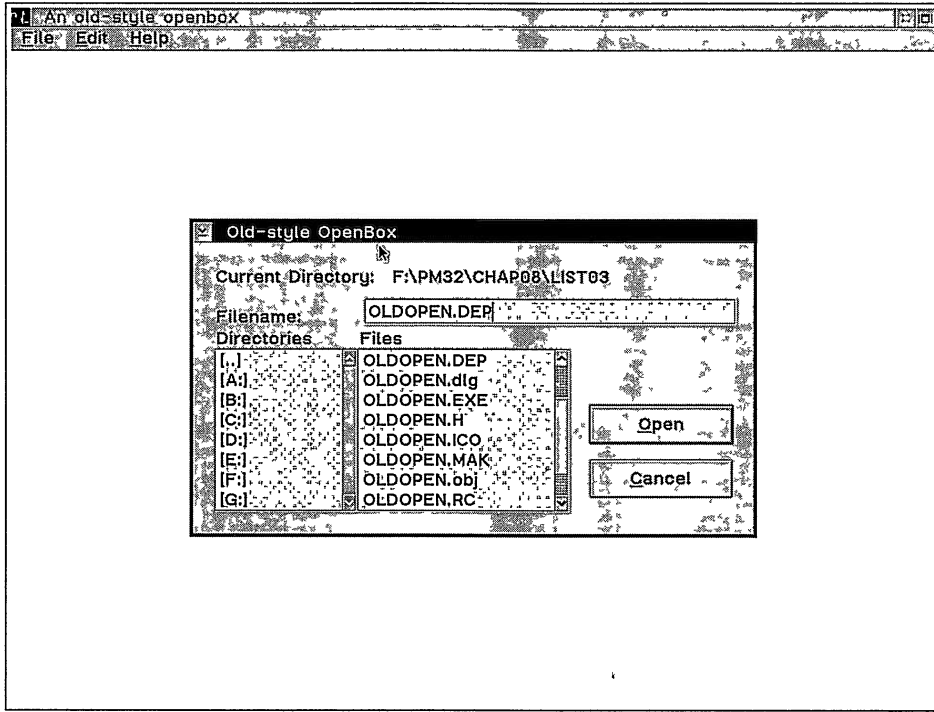
This mechanism is not limited to the first character, but extends for the whole length of text typed in by the user. So, when the name being typed in becomes `OLDOPEN.D`, the possible choices are represented by `OLDOPEN.DEF` or `OLDOPEN.DLG`, as illustrated in Figure 8.20.

To search for a text string in a listbox, send the message `LM_SEARCHSTRING`, and specify how the search is to be performed: in this case the style `LSS_SUBSTRING` is used.

```
...
while( TRUE)
{
    sPos = (SHORT)WinSendDlgItemMsg( hwnd, DL_FILELIST,
                                    LM_SEARCHSTRING,
                                    MPFROM2SHORT(LSS_SUBSTRING, sIndex),
                                    MPFROMP( buffer) );
    ...
}
```

First, the return value of `WinSendDlgItemMsg()` is checked to see that there does not exist any file name that matches the text string typed by the user in the entryfield (that's the `sPos < 0` condition). The second condition being checked determines if the involved item has an index that is lower than that identified in a previous search.

```
...
// no match, let's skip
if( sPos < 0 || ( sPos >= 0 && sPos < sIndex) )
    break ;
...
}
```



**Figure 8.20** The scrolling of the Files listbox's contents extends for the whole length of the text typed by the user.

The algorithm then goes on and retrieves the text string from the listbox, and compares it with what the user typed into the entryfield. If there is a match, then that item is placed at the topmost position in the window.

```

...
// retrieve string
sLen = (SHORT)WinSendDlgItemMsg( hwnd, DL_FILELIST,
                                LM_QUERYITEMTEXTLENGTH,
                                MPFROMSHORT( sPos), 0L) ;
WinSendDlgItemMsg( hwnd, DL_FILELIST, LM_QUERYITEMTEXT,
                  MPFROM2SHORT( sPos, sLen + 1),
                  MPFROMP( szTmp)) ;

if( strstr( szTmp, buffer) == szTmp)
{
    WinSendDlgItemMsg( hwnd, DL_FILELIST, LM_SETTOPINDEX,
                      MPFROMSHORT( sPos), 0L) ;
    return 0 ;
}
...

```

If there is no match, then the search will go on only if the first character typed in by the user is less than the corresponding character in the item just identified.

```

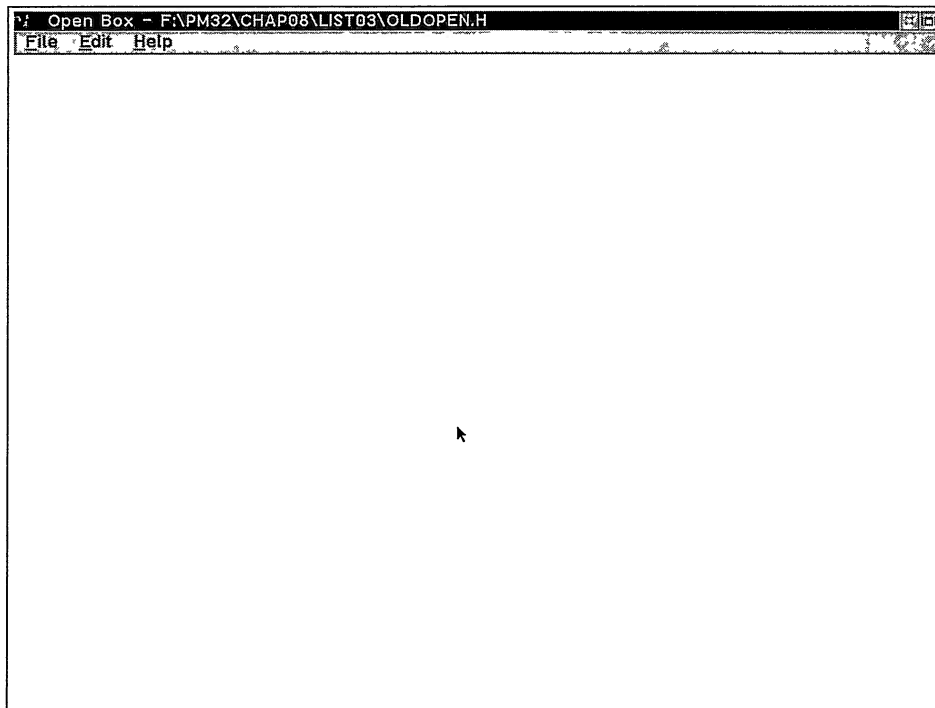
...
// check if first char in entryfield
// is bigger than first in string
if( *szTmp > *buffer)
    break ;

// set new starting position
sIndex = sPos ;
...

```

### *Selecting a File*

Apart from the technique used for selecting a file from the listbox, the application will only return its name to the calling window procedure, storing it in the memory area shared with the dialog procedure. For sake of exercise, the application then changes the contents of the titlebar by writing the file selected through the open box to the right of the Open Box title (Figure 8.21).



**Figure 8.21** The name of the selected file is reproduced in the application's titlebar.

## Predefined Dialogs

The PM API has been enriched with new functions to simplify writing applications. The designer can now use predefined dialog windows to access the file system (*openbox* and *savebox*), and even display the fonts available in the system. The positive aspect of predefined dialogs is essentially in the standardization of some parts of the application's interface, and consequently making the learning curve less steep as users get used to a consistent look. The system editor takes advantage of the predefined dialogs in all file system access operations and in font selection.

There are, however, some drawbacks. In the first place, they do not make writing code that much simpler. Predefined dialogs are based on the retrieval of members from rather complex data structures that have lots of styles for controlling customization and behavior. In the second place, the WPS style rules reduce considerably the need to design applications according to traditional schemes, still anchored to the presence of menu bars and associated drop-downs. As we will see in Chapter 13, some fundamental operations, like opening a file, saving it, or changing a font, can be implemented according to new schemes that follow the object-oriented logic of the system's interface.

## Accessing the File Systems

To implement a dialog of the type *openbox*, *savebox*, or *saveasbox*, you can follow the same design plan. The trick is all in the retrieval of the 20 members of the FILEDLG structure, with appropriate values and defines present in PMSTDDL.H.

```
typedef struct _FILEDLG
{ // filedlg
    ULONG cbSize ;
    ULONG fl ;
    ULONG ulUser ;
    LONG lReturn ;
    LONG lSRC ;
    PSZ pszTitle ;
    PSZ pszOKButton ;
    PFNWP pfnDlgProc ;
    PSZ pszIType ;
    PAPSZ papszITypeList ;
    PSZ pszIDrive ;
    PAPSZ papszIDriveList ;
    HMODULE hMod ;
    CHAR szFullFile[ CCHMAXPATH ] ;
    PAPSZ papszFQFilename ;
    ULONG ulFQFCount ;
    USHORT usDlgId ;
    SHORT x ;
    SHORT y ;
```

```

    SHORT sEAType ;
} FILEDLG ;
typedef FILEDLG *PFILEDLG ;

```

In Table 8.3 the usage of each member of the structure FILEDLG is described.

Once the members of the FILEDLG structure have been appropriately filled in, the only remaining action is that of calling *WinFileDlg()* according to the following syntax:

**Table 8.3 The Members of the Structure FILEDLG**

<i>Member</i>	<i>Description</i>
cbSize	Size of the structure.
fl	FDS_ flags to define the behavior of the dialog.
ulUser	Field available to the programmer.
lReturn	Return value from the dialogs destruction.
lSRC	Return value in case of error.
pszTitle	Title of the dialog.
pszOKButton	Title to display in the OK button.
pfnDlgProc	Entry point of the dialog's customized dialog procedure (optional).
pszIType	Text string containing the kind of filter for extended attributes.
papszITypeList	Address of a table of pointers to text strings concerning various kinds of extended attributes.
pszIDrive	Text string containing the name of the initial drive.
papszIDriveList	Address of a table of pointers to text strings pertaining to the drives to be examined.
hMod	Handle of the module from which the customized dialog template is to be retrieved if the style FDS_-CUSTOM is used.
szFullFile[ CCHMAXPATH]	Complete pathname of the selected file.
papszFQFilename	Address of a table of pointers to text strings pertaining to the selected files.
ulFQFCount	Number of selected files in the dialog.
usDlgId	ID of the <i>dialog template</i> defined in the application, which should be used in place of the standard one if the style FDS_-CUSTOM is used.
x	Position of the dialog along the X axis.
y	Position of the dialog along the Y axis.
sEAType	Type of extended attribute to associate with a file when it is saved.

```
#define INCL_WINSTDFILE
HWND APIENTRY WinFileDlg( HWND hwndP,
                          HWND hwndO,
                          PFILEDLG pfiled) ;
```

<i>Parameter</i>	<i>Description</i>
hwndP	Handle of the parent window
hwndO	Handle of the owner window
pfiled	Address of a FILEDLG structure

<i>Return Value</i>	<i>Description</i>
HWND	Handle of the dialog if the style FDS_MODELESS has been set (NULLHANDLE in case of error) or TRUE if this is a <i>modal</i> dialog

The information regarding the selected file is contained in the `szFullFile` member, while in the case of a multiple selection *openbox*, you will have to examine the contents of the whole block `papszQFilename`.

## Listing Fonts

The same logic described for `WinFileDlg()` applies to the operation of displaying the list of fonts installed in the system. In this case, though, the structure will have the name `FONTDLG`, and the function to be called is `WinFontDlg()`:

```
#define INCL_WINSTDFILE
HWND APIENTRY WinFontDlg( HWND hwndP,
                          HWND hwndO,
                          PFONTDLG pfontd) ;
```

<i>Parameter</i>	<i>Description</i>
hwndP	Handle of the parent window
hwndO	Handle of the owner window
pfontd	Address of a FONTDLG structure

<i>Return Value</i>	<i>Description</i>
HWND	Handle of the dialog if the style FDS_MODELESS has been set (NULLHANDLE in case of error) or TRUE if this is a <i>modal</i> dialog

In Figure 8.22 you can see the `NEWOPEN` application immediately after selecting the Open option. The dialog that appears at the center of the screen is a predefined *openbox*.

In Figure 8.23 `NEWOPEN` shows the *fontbox* activated after selecting the Font option from the File menu.



Listing 8.4 presents the source code of the `NEWOPEN` application and is an excellent starting point for implementing predefined dialogs in applications.



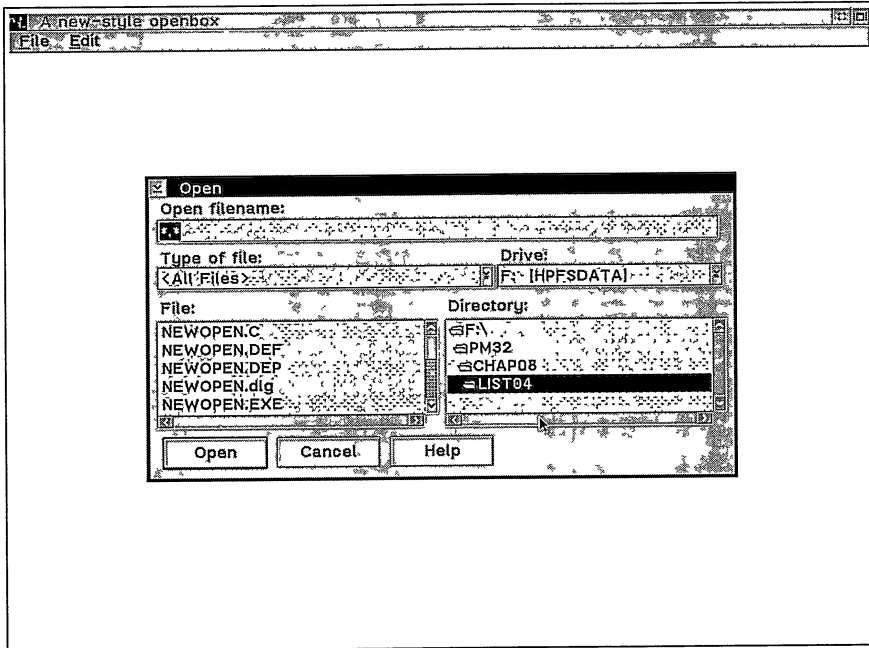


Figure 8.22 NEWOPEN takes advantage of the predefined openbox dialog to access the file system.

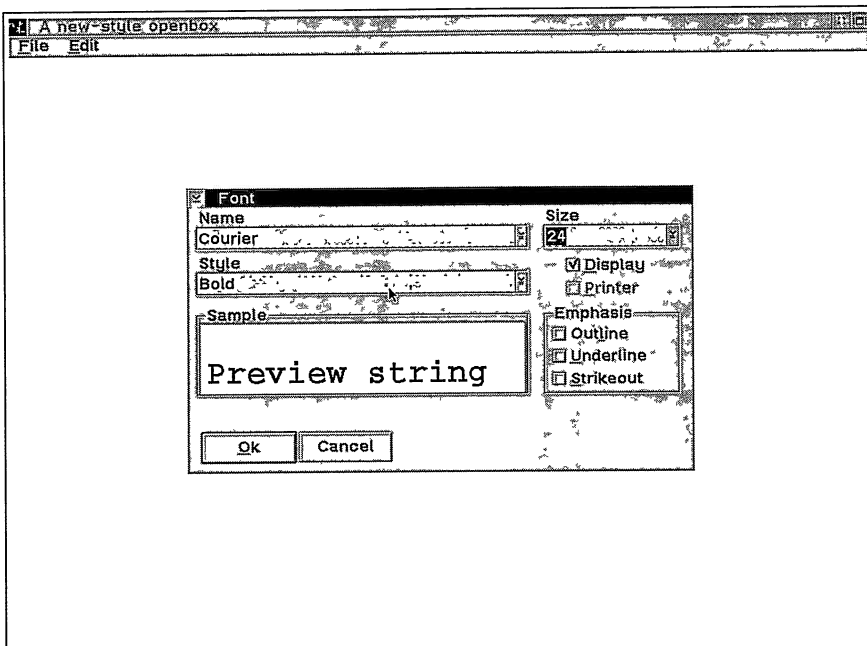


Figure 8.23 Displaying the system fonts after the selection of the Font option.

## Modeless Dialogs

Some fundamental aspects of the nature of modeless dialogs were described in the first part of this chapter. Let's summarize them briefly:

- A *template* is mandatory
- The template must be loaded with `WinLoadDlg()`
- A dialog procedure has to be written specifically for the window
- Access and handling of the modeless dialog is performed through the handle returned by `WinLoadDlg()`

Although these are the distinctive features of a modeless dialog, often the dialog procedure is not written; instead it is preferred to simply convey all messages to some window procedure that is already available. This is exactly what was done in Listing 7.6. The effect produced by `WinLoadDlg()` is that of returning to the application the handle of a window belonging to the class `WC_FRAME` inside which appear a number of controls. The handling of this window follows the traditional rules already seen in the preceding chapters. The destruction of a modeless dialog does not take place through `WinDismissDlg()`, but with the more general `WinDestroyWindow()` function, which is usually called from a *window procedure*.

### A Sample of a Modeless Dialog

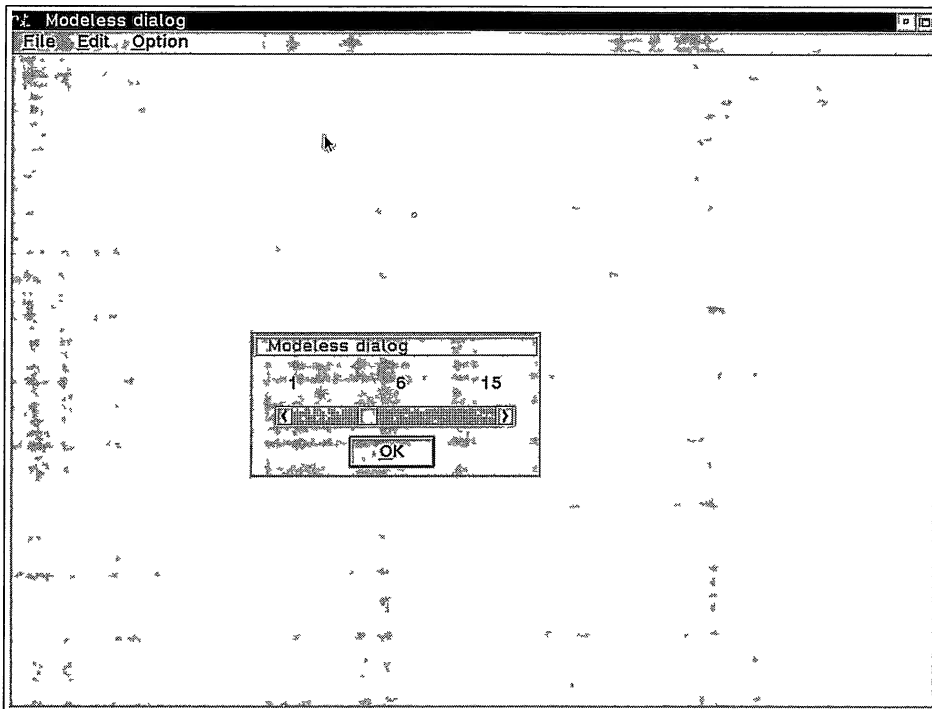


To illustrate possible use of a modeless dialog, let's once again change the color of the client window. In Listing 8.5, the scrolling amplitude of the scrollbar corresponds to the numeric values of the colors in PM. The user can change the color of the client window by acting directly on the scrollbar contained in a modeless dialog (Figure 8.24). After selecting a color, focus can be transferred to the application's main window without making the modeless dialog disappear.

The interaction between the dialog and window procedure relies on the receipt of the notification codes of the scrollbar, which are transformed into input for coloring the window. The client window's handle is passed to the dialog procedure, which is ready to color the window when receiving the `WM_PAINT` message. The modeless dialog will send the client an application-defined message (`#define WM_COLOR WM_USER`) whenever the user selects a color from the scrollbar.

```
...
WinSendMessage( hwndOwner, WM_COLOR,
                MPFROMSHORT( sPos), 0L );
...
```

The message is then processed in the window procedure, retrieving from `mp1` the numeric value of the color and then invalidating the whole window.



**Figure 8.24** The modeless dialog of this application is yet another input source in addition to menus.

```
...
case WM_COLOR:
    clr = LONGFROMMP( mp1 ) ;
    WinInvalidateRect( hwnd, NULL, TRUE ) ;
    break ;
...
```

... ..

...

...

...

...

...

...

# Developing Fast Multithreaded Applications

One of the typical features of the iAPX86 family of Intel's processors is the subdivision of the addressable space into segments. Up to, and including the 286, segments could be at most 64KB. For the 32-bit processors (386, 486, and Pentium), segments can be as large as 4GB. Furthermore, the 286, 386, 486, and Pentium provide a virtual address space that frees the operating system from the actual amount of physical memory installed in the PC. Table 9.1 summarizes the features of all Intel processors currently available.

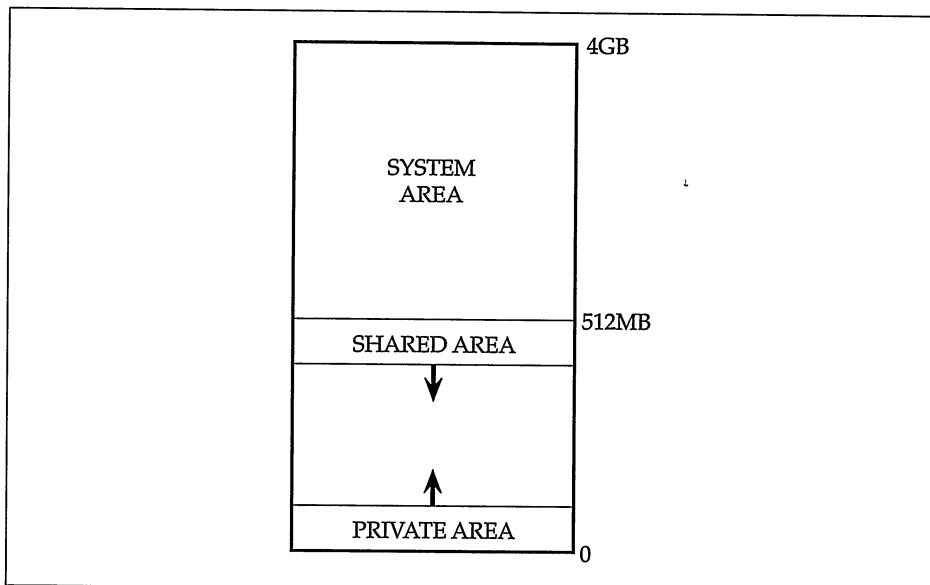
**Table 9.1 Operating Features of Intel's iAPX86 Processor Family**

<i>iAPX86</i>	<i>Mode</i>	<i>Emulation</i>	<i>RAM</i>	<i>Virtual Address Space</i>
8088	Real mode	None	1MB	Absent
8086	Real mode	None	1MB	Absent
80286	Protected mode 16-bit	Real mode	16MB	1GB
80386dx - dx2	Protected mode 32-bit	Real mode-protected mode 16-bit	4GB	64TB
80486dx - dx2	Protected mode 32-bit	Real mode-protected mode 16-bit	4GB	64TB
Pentium	Protected mode 32-bit	Real mode-protected mode 16-bit	4GB	64TB

The features of the 32-bit processors are a significant improvement over the 16-bit architecture, appreciated by system programmers as well as software developers. OS/2 2.1 can be installed only on a 32-bit Intel processor or compatible. Segmentation is an intrinsic operating mode that cannot be disabled in any way. Therefore, even the 386, 486, and the Pentium are segmented processors. This, however, does not imply that OS/2 must be segmented as well. As a matter of fact, the implementation of OS/2 on a 32-bit processor is unusual. The concept is very simple: A single 4GB segment gives an address space that is vastly superior to that attainable in the 16-bit mode, and is large enough to contain both the operating system as well as the applications.

Therefore, even if the 32-bit Intel processors are segmented, OS/2 “sees” just one flat address space of 4GB. For this reason, memory management is typically linear, and no longer segmented as in the past. The 32-bit offset is basically the only source of information to calculate a memory address passing through the paging algorithm.

It follows that the theoretical limits of 32-bit processors are used only partially by OS/2 and other similar operating systems. For compatibility reasons, with older 16-bit OS/2 applications, at the current stage the amount of memory that can be addressed directly by any 32-bit OS/2 process is “only” 512MB, although this limitation will soon be removed. This block contains even the memory space shared between processes. Each single running task in OS/2 2.1 can access up to 512MB of address space, which is conceivably the amount of RAM that could be physically installed in a computer (although, more realistically, the best-equipped machines are currently limited to 20–30MB). Figure 9.1 depicts a scheme of memory in OS/2 2.1.



**Figure 9.1** Memory scheme of OS/2 2.1 systems.

## Memory Allocation

The allocation of a block of memory is no longer based on the concept of the segment, but on that of the page. 32-bit processors also have a paging unit of memory, a tool that allows optimized access to a large memory area. A page in OS/2 is a block of 4.096 contiguous bytes, and is the minimum amount of memory that can be allocated. Even if you need only two bytes, you will have to allocate a whole page. Don't be deceived into thinking that this is a waste: The size of a page is so small with respect to the processor's capabilities that this is no problem. Furthermore, the adoption of the page as the minimum chunk of allocable memory provides considerable advantages when swapping memory block to the hard disk. Another feature is the automatic initialization to zero for each allocated page.

The linear addressing of memory, released of the limitations and constraints of the old segments, makes things easier. The first change is the disappearance of memory models. You will no longer have to choose between a Small rather than a Medium or Large memory model. Just compile, and you're done. Consequently, even pointers are of one kind, and always four bytes large.

If you then need to access to a large area of memory (maybe some megabytes!) you just need to call *DosAllocMem()* and the trick is done!

```
#define INCL_DOSMEMMGR
APIRET APIENTRY DosAllocMem( PPVOID ppb, ULONG cb, ULONG flag);
```

<i>Parameter</i>	<i>Description</i>
ppb	Address of the pointer to the memory block
cb	Size of the memory block
flag	Attributes of the memory block
<i>Return Value</i>	<i>Description</i>
APIRET	Success or failure of the operation

Compared with the memory allocation techniques in 16-bit environments, there is one more important difference. The memory allocated by *DosAllocMem()* can effectively be used through a pointer only if the flag *PAG\_COMMIT* has been set. *Committing* is what actually makes available the set of pages requested by the designer. The separation of the request from use (allocation and committing) grants that you will always be able to have large memory area without overloading the system with oversized blocks. In fact, it is always possible to commit new pages at any later moment by calling *DosSetMem()*.

In the 32-bit world of OS/2 there is yet another peculiarity regarding memory allocation operations. Once a block has been allocated, the only operation that is allowed on it is its destruction (in addition to the dynamic commitment). There is no way of reallocating a block; but this is no problem, thanks to the delayed commitment techniques.

## Memory Management

The Memory Manager of OS/2 resorts to three distinct techniques in dealing with and solving any problems derived from overcrowded memory due to multiple simultaneously running applications:

- Moving
- Discarding
- Swapping

The first technique consists of changing the position of memory blocks within the physical RAM present in the system. This can prevent excessive fragmentation of the residual memory into several portions too small for the system's needs. The compacting of these free areas can create one block of contiguous RAM large enough to contain a new application launched by the user. The movement of memory blocks is a very fast operation, and does not require any kind of effort on part of the operating system, as it is supported directly in hardware. However, the rearrangement of the memory map will not always suffice to fulfill the needs of the system when even larger blocks of free memory are requested.

The second technique employed by OS/2's Memory Manager is that of free memory from those objects that are *discardable*. When defining resources like menu templates or dialog templates, you can use the memory option DISCARDABLE to define the behavior of a menu or a dialog window once it has been loaded into memory. A discardable object will allow OS/2's Memory Manager to free the space it occupies whenever the need for more memory arises. In order for the assignment of the attribute of DISCARDABLE to be correct, it is necessary that it is associated with a read-only resource. The action performed by the Memory Manager on a discardable object is destroying it and freeing the RAM it held. This will be possible only if that resource's contents do not get changed after being loaded in the system. The code segment and resources like a menu template and a dialog template fall into this category. The system maintains a table of objects marked *discardable* in order to identify those least recently used (LRU strategy), and that have not been locked. Furthermore, it also maintains a reference to all free discardable segments to specify if they are present in memory or released due to a previous request for free memory. When the application needs to access again an object that was previously discarded, it will automatically load it into RAM in a way that is completely transparent to the application.

The third technique employed by the Memory Manager when there is little free memory, is that of making a copy on disk of some portions of RAM (*swapping*). Of the three techniques used by the Memory Manager, this is the slowest, since it implies the operation of hardware devices that respond in mean access time expressed in milliseconds, compared to the nanoseconds of RAM. However, it is through *swapping* that OS/2 systems can continue to operate even in conditions of extreme memory scarcity.



## Segmented Applications

The three techniques described in the previous paragraph are used and handled by the Memory Manager of OS/2 when loading and executing processes. The software designer has a very limited duty in all this—just to decide which objects should be of the *discardable* type in the resource file or when allocating new memory blocks.

Thanks to the implementation of OS/2 on 32-bit processors, each program is made up of just one code segment and one data segment (respectively CS and DS). The term “segment” in the linear world of OS/2 has nothing to do with the equivalent concept in the 16-bit world. It means more generally an “object.” The linker of OS/2 systems is capable of generating an executable with subportions with different attributes governing the time at which they are loaded into memory and how they are subsequently managed. In the past, these subportions coincided with the segments of the processor, and for continuity, the same name has been kept, albeit with a different meaning. Therefore, it is possible to generate code that optimizes its impact on the system’s memory by delaying in time the moment of its being loaded into RAM. To achieve this, the programmer will need to subdivide the code into different source files, generating a corresponding number of object modules. The definition of “segmented” executable refers thus to the different behavior taken by the linker with respect to the single object modules that make up the executable. Some modules will be loaded into RAM as soon as a process is activated (PRELOAD); others will remain on disk until they will be involved and requested directly by the application (LOADONCALL) as shown in Figure 9.2. This same behavior can be extended to some resources specified in the resource file, as described in Chapters 6 and 8 (menu templates and dialog templates).

### Code Segment and Data Segment

Each PM application consists of one single code segment and one single data segment. This feature derives from the implementation devised by IBM’s designers for the

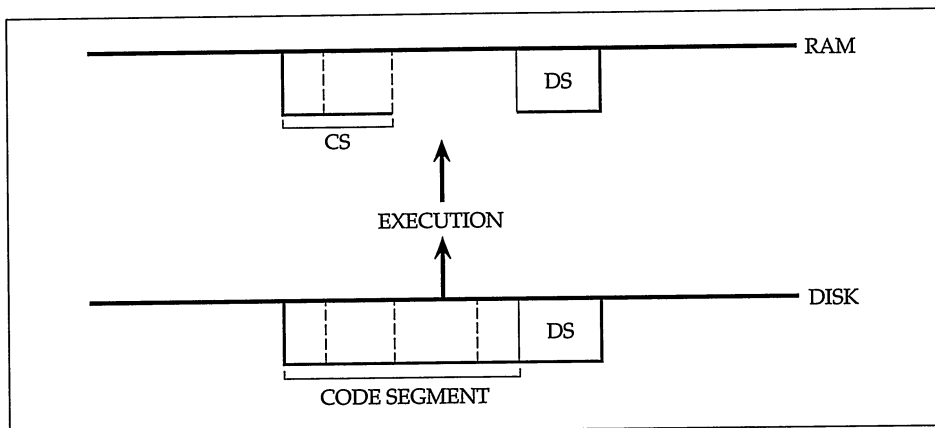


Figure 9.2 Operating scheme of the loading of a segmented OS/2 PM application.

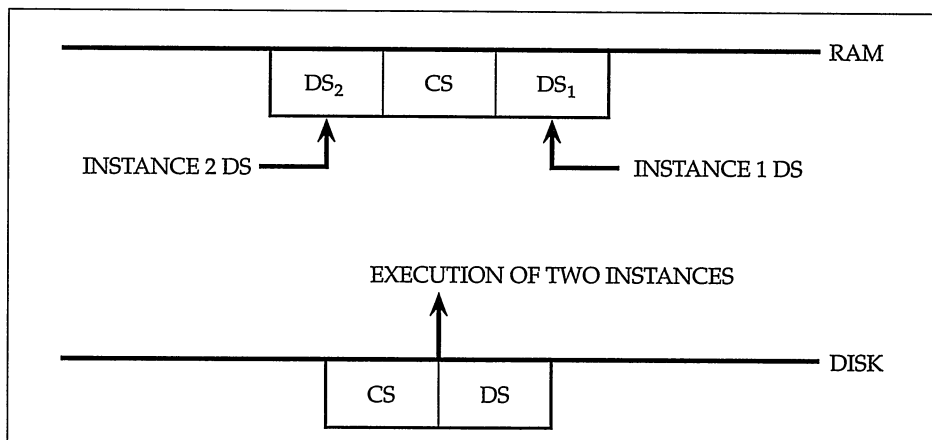
32-bit Intel processors. In the 16-bit world it is customary to reason in terms of memory models that differ in the kind of pointers, and number of code and data segments, each at most 64KB. You will have to forget all about this in OS/2 2.1. At the current stage, OS/2 applications have access theoretically to a flat address space that is at most 4GB, but limited in practice to 512MB for compatibility reasons with the older 16-bit code of version 1.x (this limit will be removed in forthcoming versions). The maximum size of the code segment and of the data segment is in any case several magnitudes greater than the past limit, and no longer creates a bottleneck even for the most complex and largest applications. A shared memory area can be 64MB, and will have to be subtracted from the limit of 512MB. Each process thus has a practical address space of approximately 448MB.

### Executing Several Instances

The problem of executing several instances of the same application has been dealt with in Chapter 4 as far as windowing issues were concerned. Now we will see what happens in the system with the code and data segments that characterize the executable.

A criterion followed to minimize the occupation of memory is that of sharing among the various instances the code segment of the application. In the systems that operate in *protected mode*, all code segments are by definition read-only, and as such will never be subject to any kind of modification during their execution. Any attempt to change their contents will be rejected by hardware, which will generate a *General Protection Fault*.

When an application is executed, the system loader loads all its code and data segments that have been declared with the directive `PRELOAD`. In any subsequent execution of other instances, the program will load only the data segments, and not the code segments already present in memory. Figure 9.3 illustrates this.



**Figure 9.3** The use of the code segments and data segments by different instances of the same application.

In order for every instance to have private data segments, it is necessary that the `DATA` directive be completed by `MULTIPLE` to indicate multiple copies of data segments. The default values assumed by the linker are `LOADONCALL` both for code segments as well as for data segments.

## *Producing a Fast Executable*

Most often, in an OS/2 application it is possible to identify portions and/or options of the menu bar that are used less frequently than others. It is easy to guess that the `File:Open` option is used frequently in any PM application. Less frequently will the user select `Product Information`, since the information returned is static and seldom used in the application.

The `Product information` option generally displays a dialog window; that implies the presence of a dialog procedure and a dialog template. Loading into memory these two objects directly when the application is running will often mean occupying memory space without there being any real and tangible reason for it. It would be much better to defer the loading into memory to the moment when it is truly needed. Both of these objects can be declared with the `LOADONCALL` memory option. As far as the dialog template is concerned, it is sufficient to assign the option directly in the resource file, as in the following example:

```
DLGTEMPLATE 256 LOADONCALL MOVEABLE DISCARDABLE
{
    ...
}
```

For the dialog procedure, you have to follow an approach that implies the implementation of the following conditions:

- Writing of the dialog procedure in a separate file with respect to that of the actual application
- Compiling the source code
- Assigning the attribute `LOADONCALL` to the code segment containing the dialog procedure of the `Product Information` option
- Linking the application

For the first step, it is simply a matter of applying the common programming techniques in the C Language. The subdivision of code into distinct modules makes it simpler to write and maintain the code, as it allows one to concentrate on the program. When compiling this source, there should be no particular flags for the actual source code. To assign the attribute `LOADONCALL` to the segment corresponding to the object module containing the dialog procedure of the `Product information` option, you have to act at the level of the module definition file (`DEF`).

## The Structure of the DEF

The realization of a segmented application is the outcome of the assignment of directives in the DEF file, regarding the linker's results. In writing the module definition file, you define by means of the directives CODE, DATA, and SEGMENTS the features that pertain to their loading and management in memory. According to the syntax and functioning of the linker 2.00, it is necessary to indicate the directives CODE and DATA with the following default values: PRELOAD MOVEABLE and DISCARDABLE.

This means that all code segments and data segments of an application will be loaded into memory when the process starts execution. With the directive SEGMENTS you can discriminate between all segments of the application, code or data, by giving them a behavior different from what is indicated with CODE and DATA.

The identification of segments with features different from that indicated by CODE and DATA will force you to state in a precise way the name of the code and the data segments you are interested in. To achieve this, you must know the *module name* of each object file that you want to treat as a separate segment with respect to the rest of the application (Figure 9.4).

The module name corresponds to the name that is present internally in each OBJ file, and sought by the linker when it generates the executable. The internal name of an object module is, by default, equal to the name of the source file at the file system level, with the addition of CODE32 or DATA32, respectively, for code segments and for data segments.

On the basis of the previous example, the internal name of the object file PRODINFO.OBJ becomes PRODINFOCODE32: This is the information that is specified

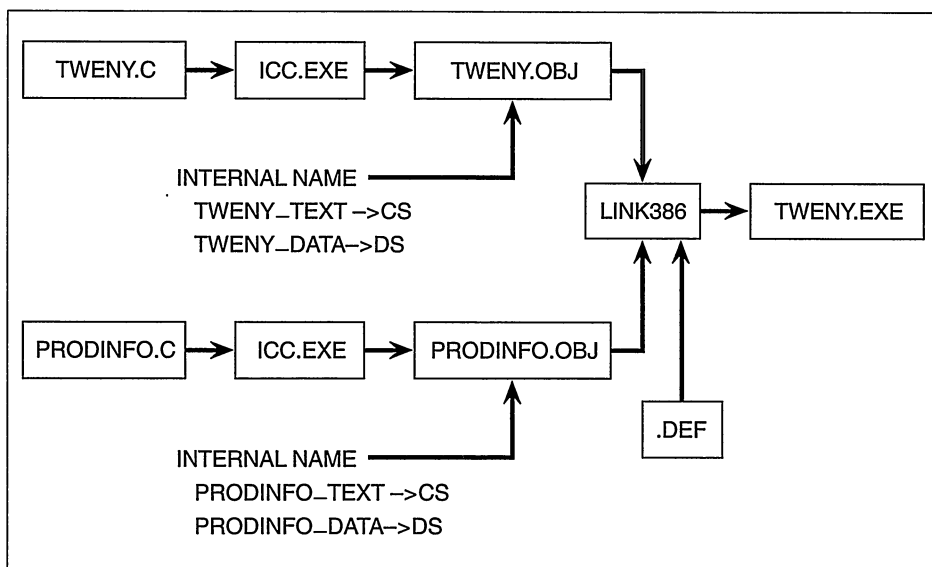


Figure 9.4 The creation of distinct code and data segments for subsequent handling by the system's linker.

correspondingly to the SEGMENTS directive. The linker explicitly assigns the loading and memory management options to the segments declared in the SEGMENTS directive:

```

...
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE DISCARDABLE MULTIPLE

SEGMENTS
  PRODINFOCODE32 LOADONCALL MOVEABLE DISCARDABLE
...

```

The code segment PRODINFO\_CODE32 is marked with LOADONCALL, and therefore will not be loaded into memory when the application is running until the user selects the option Product information. As this is an object of the type *discardable*, it will be released from memory when the system subsequently receives a request for available RAM.

The rule governing the use of CODE, DATA, and SEGMENTS does not imply the exclusive use of the PRELOAD option for the first two directives. The criteria to be followed is that of declaring with CODE and DATA the greatest possible number of segments, and then distinguishing them with SEGMENTS. If an application has a number of segments of the type LOADONCALL that is greater than the number of the type PRELOAD, it is convenient that this attribute be specified correspondingly to CODE and/or DATA. In the next chapter we will examine in greater detail the syntax of the DATA directive, when dealing with the development of *dynamic linking libraries* (DLLs) for OS/2.

## Segmentation Rules

While the protected mode of the Intel iAPX86 processors delivers an addressing space that is much larger than that of real mode, often the amount of physical RAM installed in the system turns out to be inadequate to contain very large applications. The technique of segmentation gives the software designer a simple means for overcoming this problem, at least partially. The advantages of this solution are many:

- Shorter times for loading an application into memory
- Shorter waiting times for the user during the initial activation operations of the program
- Ability to work even on systems with a limited amount of memory
- Better contribution to multitasking the system

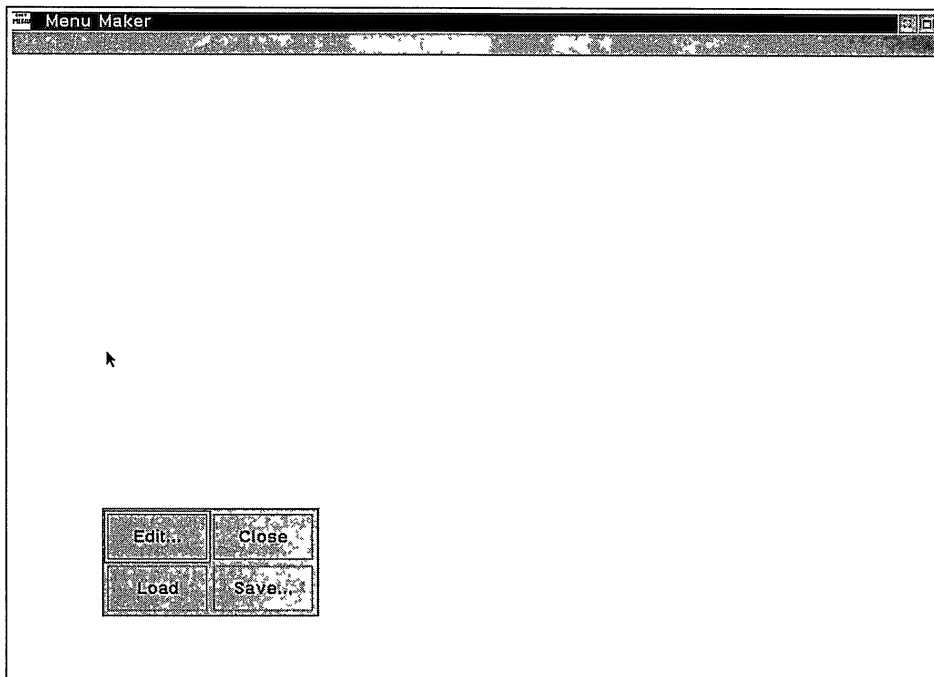
In general, all options that are selected less frequently in the application can be transformed into code portions resident in distinct segments and marked with LOADONCALL. The overhead during the design phase is minimum; the only addition is a relatively greater complexity in the module definition file. In the next section we will examine the structure of an application called Menu Maker, a utility for generating menu templates. The relative complexity of the program, and its operations being strictly dependent on the choices made by the user, make it an excellent way to practice code segmentation.

To this end, the analysis of the source code with the development environment's browsing options is extremely rewarding for recognizing functions that are interrelated. Different from what was stated in the previous example regarding the Product Information option, it is not mandatory that a segment contain only one code function. Rather, it is advisable to collect in a single segment a greater number of functions, provided they are closely related.

---

## A Menu Editor

The OS/2 2.1 IBM Toolkit contains some tools for automatically producing dialog templates, icons, and fonts. DLGEDIT.EXE will help you in writing a dialog window, as it allows you to draw the window and the controls it contains directly on the screen. The DLG file produced is a simple script file that summarizes all startup directives for reproducing the window drawn on the screen. The Borland compiler provides a more complete resource manager that also has a tool for producing menu templates. The market offers many other tools specifically designed for creating menu templates. What is interesting about this example is not the functionality delivered by it, but that it allows you to deal with some rather obscure aspects of the menu API of PM. In Figure 9.5 you can see the MENU MAKER utility.



**Figure 9.5** The application MENU.EXE allows you to generate a menu template by drawing it directly in a PM application.

## The Interface of Menu Maker

The Menu Maker application might look unusual; the program's window is maximized and does not have either a restore or a minimize icon; an empty menu bar is displayed; the lower portion of the client window accommodates a valueset with four cells.

## The Maximized Window

Creating a maximized window in OS/2 PM is not a difficult task due to the specific style `WS_MAXIMIZE`: the simplest solution is to set this style directly in the call to `WinCreateStdWindow()`. Unfortunately, this way of writing the application does not produce the desired result. To bypass this inconvenience, you will have to create the window with no display option, and delegate the operation and the maximization directly to the function `WinSetWindowPos()`:

```

...
ULONG flFrameFlags = FCF_WPS & ~FCF_ACCELTABLE ;

hab = WinInitialize( 0 ) ;
hmq = WinCreateMsgQueue( hab, 0L ) ;

WinRegisterClass( hab, szClassName,
                  ClientWndProc,
                  CS_SIZEREDRAW, 0L ) ;

hwndFrame = WinCreateStdWindow( HwndDesktop,
                                0L,
                                &flFrameFlags,
                                szClassName,
                                szWindowTitle,
                                WS_CLIPCHILDREN,
                                NULLHANDLE, RS_ICON,
                                &hwndClient ) ;

// show maximized window
WinSetWindowPos( hwndFrame, HwndTop,
                 0, 0, 0, 0,
                 SWP_MAXIMIZE | SWP_ACTIVATE | SWP_SHOW ) ;

...

```

## The Empty Menu Bar

Beneath the titlebar there appears a menu bar with no top-level menus. Among the *frame creation flags* declared in the `WinCreateStdWindow()` function there will also appear `FCF_MENU`: The menu template is thus very simple, as it defines only one top-level menu item:

```

...
MENU RS_MENU
{
    MENUITEM "", MN_FAKE
}
...

```

Since it has no text, the menu item with the ID `MN_FAKE` will not display anything on the screen. However, you can still perceive its presence by pressing the left mouse button over the left extreme of the menu bar. Its selection will not cause any effect in the application. By specifying the style `MIS_BUTTONSEPARATOR`, the top-level gets right-aligned on the menu bar, instead of left-aligned—in this case its detection is somewhat more difficult. As shown in Figure 9.6, the menu bar is a window of the class `WC_MENU`.

As discussed in Chapter 6, there is no specific data structure for the class `WC_MENU`, even if the toolkit's documentation proposes a kind of structure onto which you could base the creation of a window belonging to the class `WC_MENU` directly through `WinCreateWindow()`. Listing 6.10 resorts to that technique for creating a window of the class `WC_MENU`. In this case, instead, the creation of a top-level menu and of the menu items of the drop-down menus is performed dynamically under the direct control of the user of the application, and not under the control of the programmer. In any case,

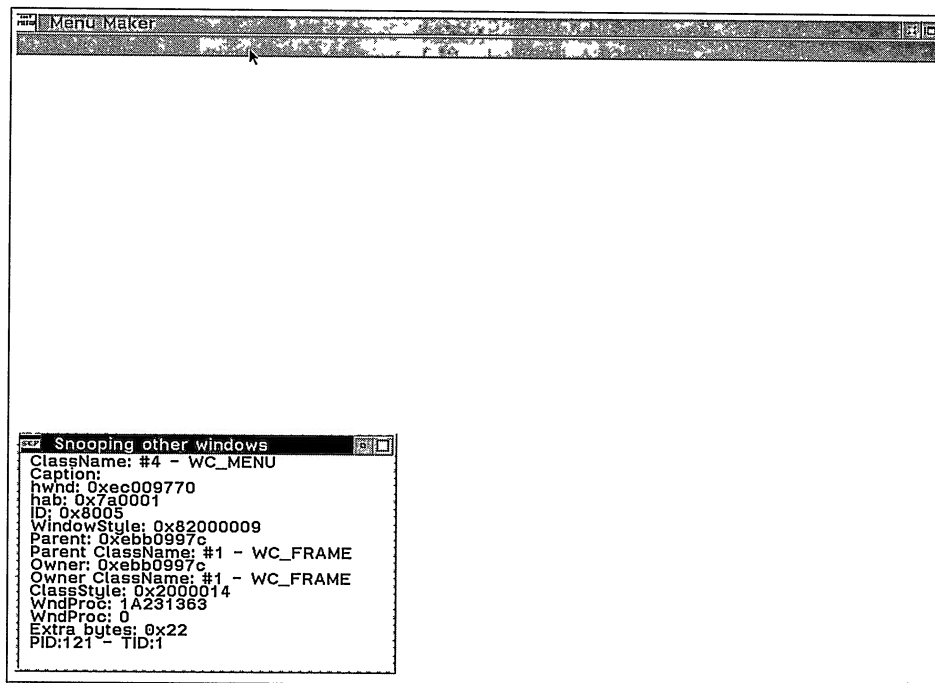


Figure 9.6 SNOOPER proves that the menu bar is a window of the class `WC_MENU`.



it is not possible to create a menu bar without at least one menu item: The loading of a menu template from the resource file is the simplest and most convenient solution.

## The Control Panel

A window of the class `WC_VALUESET` appears in the lower portion of the application's client window, and is the sole means of communication between the user and the program. The `Edit...` cell mimics with ellipses the look and feel of menu items of the extended command type. Its selection will display the modeless dialog shown in Figure 9.7, which is fitted for generating the menu structure.

Shortly, we will examine how the dialog works. The presence of the dialog is often subsequent to the loading of a template that can be obtained by pressing the `Load` cell, or by producing a new menu structure. With `Close`, the user can abandon the program, while with `Save`, the user can generate an image on the disk of the menus created with the application.

## The Application's Logic

Producing a menu template means generating a resource of the type `MENU`, defining its IDs, and creating an accelerator table. All of these aspects are dealt with in `Menu`

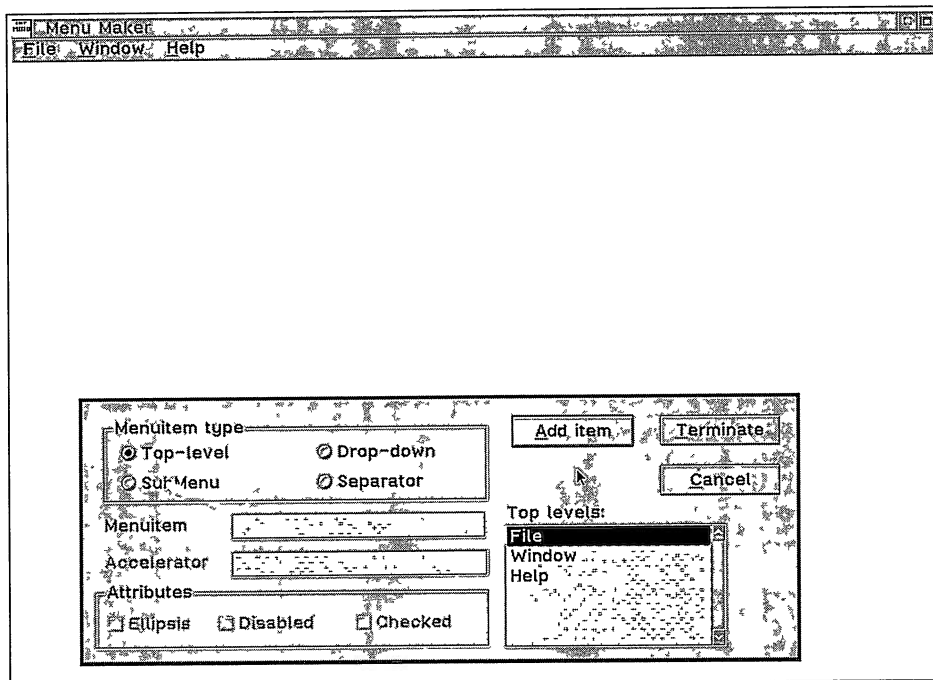
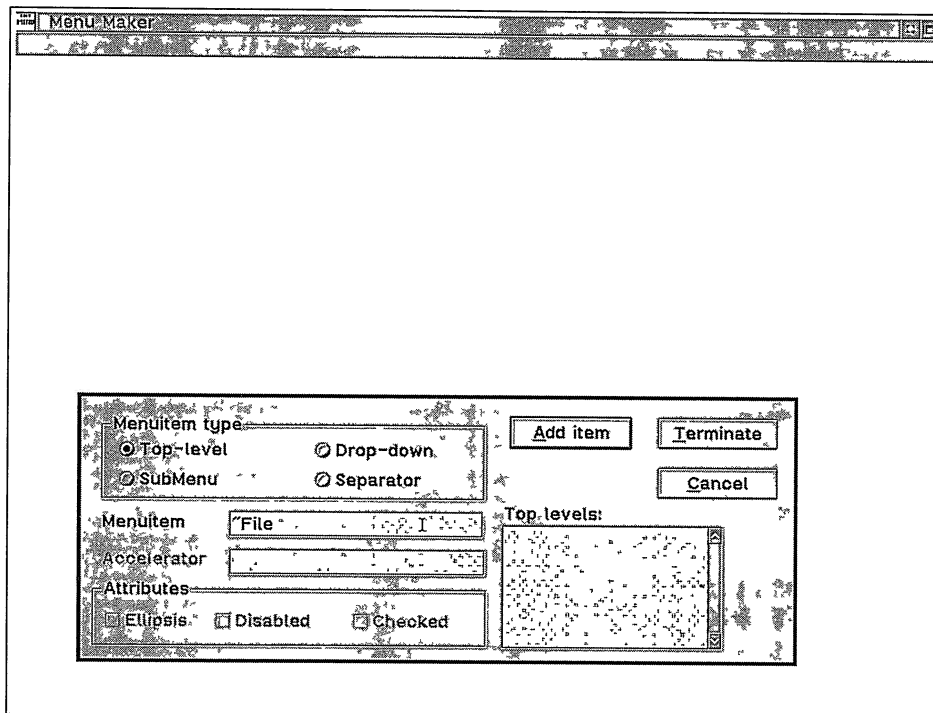


Figure 9.7 The setting of the top-level menu and of the drop-down menus is performed by selecting among the options presented in this modeless dialog.

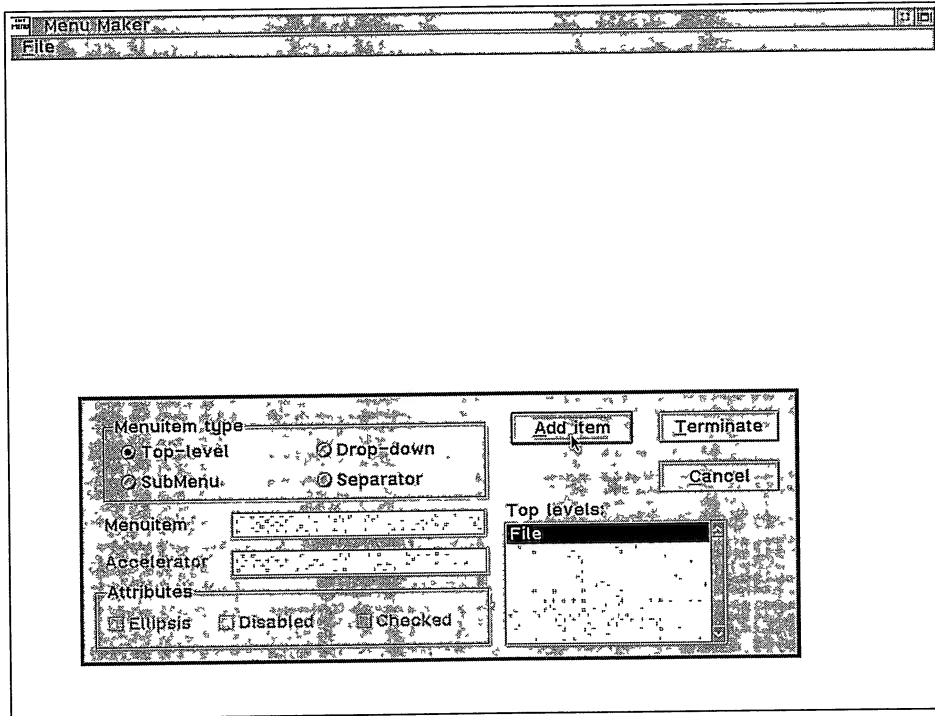
Maker. When generating a template, the operation is controlled by the modeless dialog displayed with the Edit... cell. Initially, the only option allowed is that of generating a top-level menu (Figure 9.8).

## Defining a Top-Level Menu

The user can input in the specific entryfield the text that is to appear in the menu bar. The application sets no limit whatsoever on the inserted text: Any combination of upper- /lowercase letters, as well as the definition of the underlined letter (mnemonic character), can be used. To implement this, it is convenient to create a specific function that forces the first letter to be a capital and the rest in lowercase. Furthermore, the strings of text of top-level options can contain only letters of the alphabet. The check for the mnemonic code can be as simple as checking for the presence of a tilde, or with a much more sophisticated algorithm that would even check if the selected letter were already used by some other top-level. The style attributes, represented in the dialog as check boxes, are all disabled, as is the entryfield pertaining to the insertion of an accelerator combination. This choice is imposed by the CUA rules. After inserting the text of the top-level, the user must press the button Add Item in order to insert the new item into the menu bar (Figure 9.9).



**Figure 9.8** The first operation to perform when generating a menu is that of defining one or more top-level menus.



**Figure 9.9** Insertion of the File top-level in the application's menu bar.

The choice of a modeless dialog is justified in this portion of the program. In fact, as you can see in Figure 9.7, the modeless will lose focus when it is transferred to the menu bar. The transfer of focus between the two windows allows you to check directly during the writing of the menus as to how they actually appear.

The insertion of a top-level menu that does not cause a drop-down to appear is a possible situation in PM, although it is advised against in the CUA specifications (it is preferable to transform the item into a menu item, rather than taking up space in the menu bar). Once a top-level has been created, the user defines the items to be inserted in the associated drop-down menu or defines yet other top-level menus. Each top-level defined by the user is accumulated in a specific listbox presented in the modeless dialog. This allows the user to select into which top-level menu to insert the menu items of the associated drop-down (Figure 9.10).

## *Defining a Drop-Down Menu*

After having defined a top-level menu, the four radio buttons in the dialog are activated—top-level, drop-down, submenu, and separator. In general the user will focus on the drop-down radio button. The text is always defined in the entryfield called MenuItem, while even the entryfield for defining the accelerator now gets activated, as well as the check boxes (Figure 9.11).

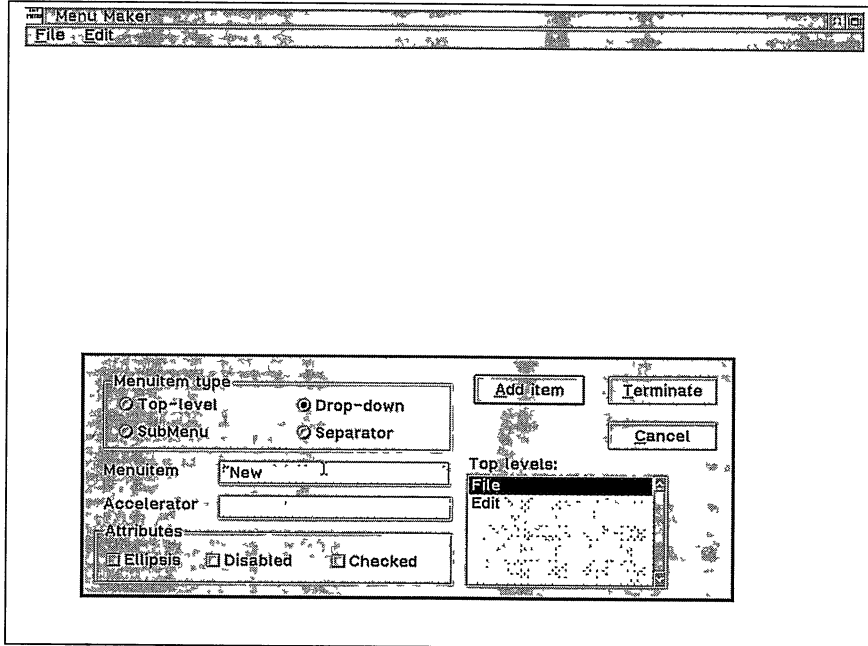


Figure 9.10 In the definition of drop-down menus, Menu Maker allows the user to select the associated top-level menu.

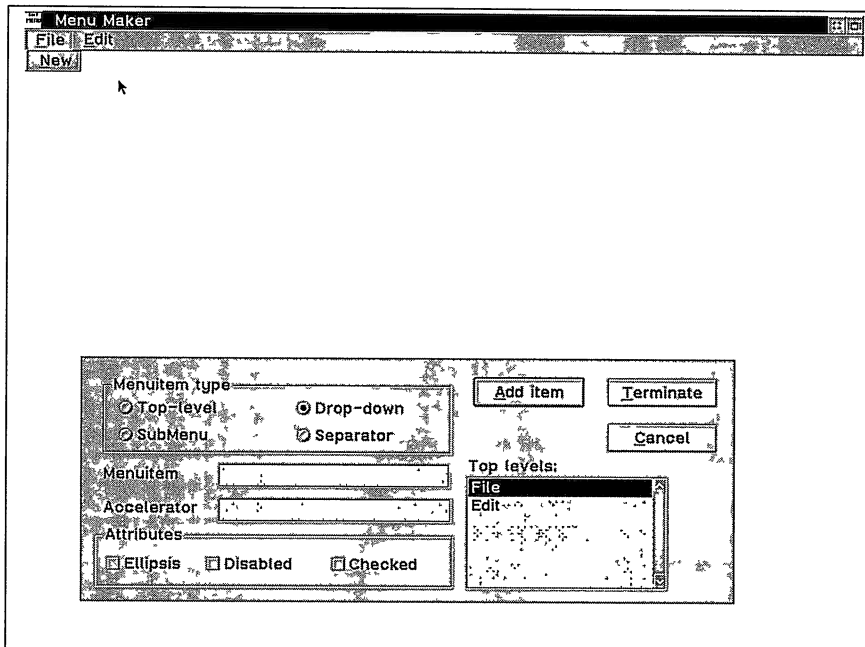


Figure 9.11 Insertion of the text pertaining to a drop-down menu.

When inserting the text describing the accelerator, the application does not perform any kind of control. The best syntax involves at most two keys for the accelerator combination, based on one or two virtual keys (CTRL, SHIFT, ALT, or Fxx) possibly followed by a letter of the alphabet.

When the first drop-down of a top-level is defined, it is necessary to change its nature: This operation is equivalent to declaring a top-level with the SUBMENU directive rather than with MENUITEM. The operation is not difficult, but it obviously requires you to have the handle of a window belonging to the class WC\_MENU. The application must provide it, and then fill it in with the text defined by the user. In the next paragraphs we will examine in greater detail the nature of the code implementing each single operation of the application.

## Defining a Separator

The operation of inserting a horizontal separator bar in a drop-down is relatively simple. When the corresponding radio button is selected, the text SEPARATOR will appear in the entryfield accommodating the menu item text (Figure 9.12).

The only quirk in the implementation of this function is in defining the rules for inserting a SEPARATOR in a drop-down. It is good to check that the bar never gets positioned at the extreme ends of a drop-down, or that it is present two or more times

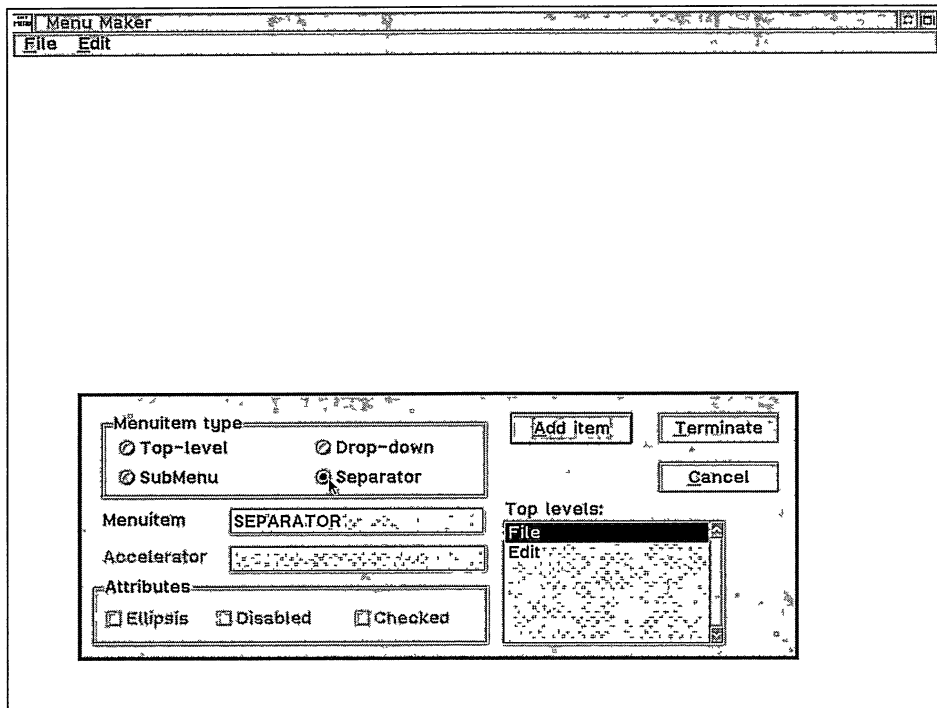


Figure 9.12 Inserting a horizontal separator bar in a drop-down.

in a row. It is possible to avoid this at the top of the window or inside a drop-down, but it cannot be checked for at the end of the list. Only the user can make sure that that does not happen there.

## Defining a Submenu

The CUA specifications are very clear concerning the creation of drop-down menus at any level beyond the first one. Although the operation is possible (since the resource compiler does not impose any practical limit on the number of concatenated drop-downs), it is nonetheless advisable to make all possible efforts to avoid it, for ergonomic and functional reasons regarding the user interface. The Menu Maker utility allows you to define a second-level drop-down, but not any one beyond the second level (which isn't acceptable in a real-world application). This means that the radio button Submenu must be handled with special care. Once you have specified an item of type SUBMENU (Figure 9.13), the corresponding radio button gets disabled, and the user will be allowed to define only a drop-down.

As before, it is not permissible to specify a SEPARATOR as the first item in a new drop-down. To inform the application that the end of the second-level drop-down has been reached, the user must select the Submenu button once again. This behavior establishes the moment when the definition of a second-level drop-down starts and

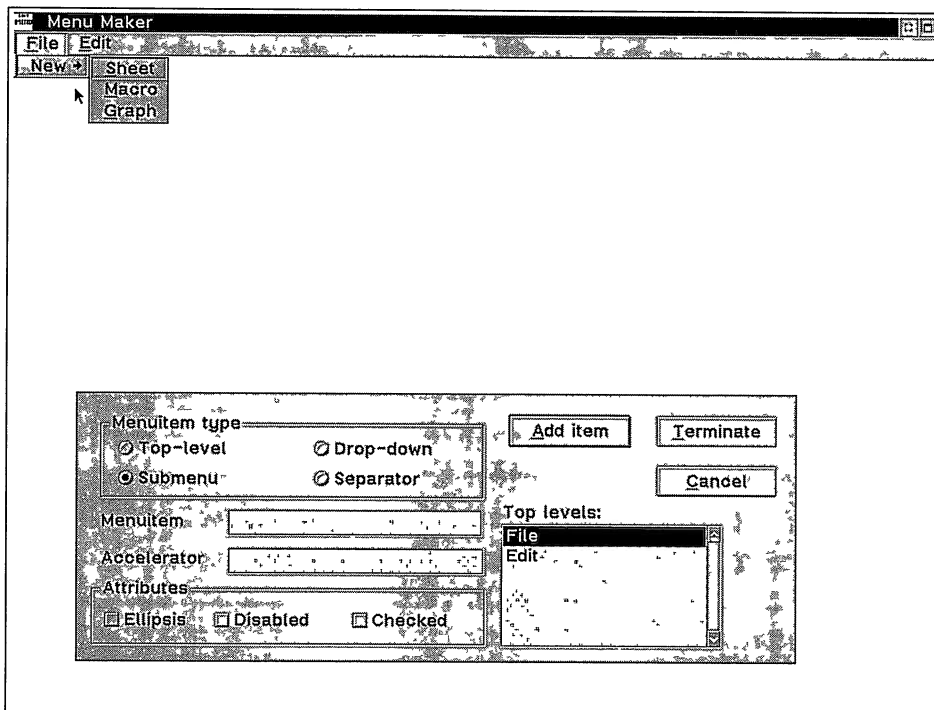
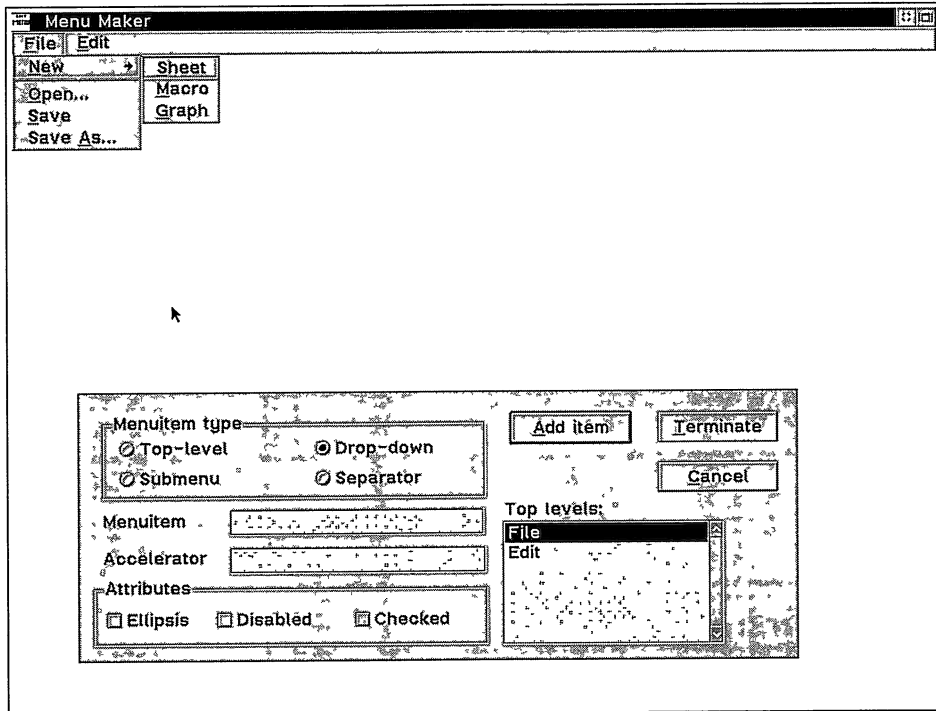


Figure 9.13 Definition of a second-level drop-down in Menu Maker.



**Figure 9.14** With Menu Maker, the menu structure of a PM application can take articulate forms with the presence of second-level drop-down menus.

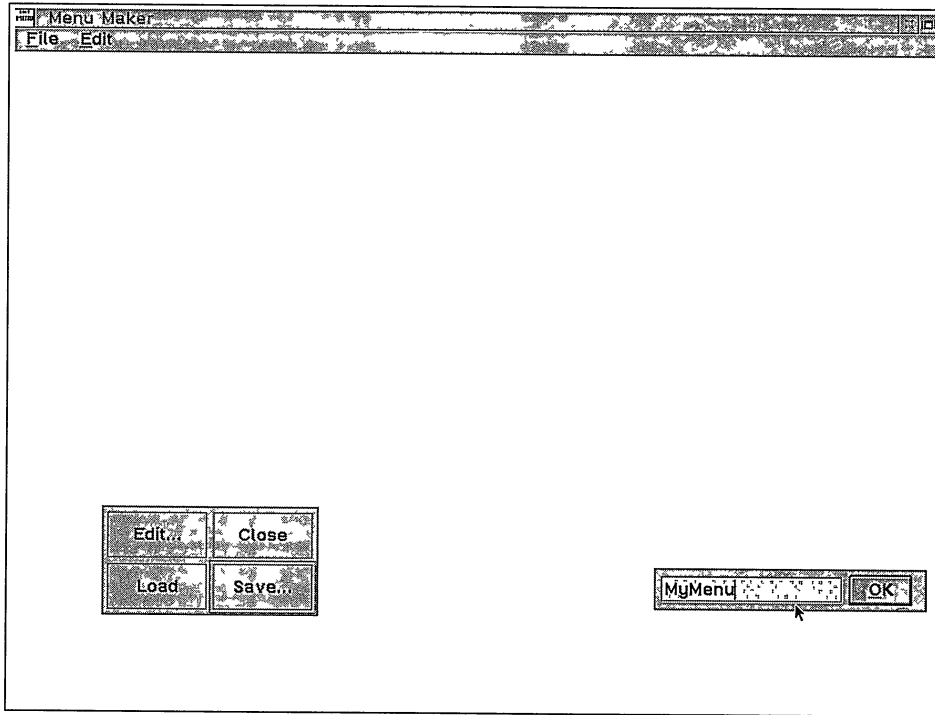
when it ends, thus granting the application access to the appropriate menu windows. In Figure 9.14 you can see the structure of the menus created with Menu Maker which contain second-level drop-downs.

## *Saving the Template*

Once the menu structure has been defined, you can abandon the modeless dialog by selecting the Terminate or Cancel button. The generation of the files containing the description of the menu window takes place when you select the Save button. A small modal dialog will appear on the screen, asking for the name to assign to the file containing the menu template, the accelerator table, and the definition of the IDs. The extensions used are, respectively: MNU, ACC, and HHH (Figure 9.15).

When writing the resource file, you then only need to establish in the RC a reference to these files in order to have them compiled. Here's an example:

```
...
rcinclude TWENY.HHH
rcinclude TWENY.MNU
rcinclude TWENY.ACC
...
```



**Figure 9.15** To save the menu structure, the user must assign a name to the set of files generated by Menu Maker.

It is mandatory that the specific menu template header file be declared before the one with the extension MNU, because it contains the necessary defines. In Figure 9.16 you can see the MNU, HHH, and ACC files generated by the Menu Maker utility.

### *Loading a Menu Template*

To transfer into memory the contents of the files MNU, HHH, and ACC, you cannot take advantage of any of the services offered by OS/2's API. The application must therefore implement a specific parser for the syntax of the three files in order to interpret the information therein contained. This solution is extremely flexible, since it also acts on *any* menu template generated with Menu Maker. It would have been much easier to simply retrieve a menu template directly from an EXE module, and let OS/2's API perform all the loading operations. Before implementing this algorithm, it is necessary to analyze in great detail the scenario we have to deal with. The templates generated by Menu Maker will eventually be inserted in the resource file associated with some application other than Menu Maker. The function *WinLoadMenu()* provides a parameter for identifying, by means of an appropriate handle, the module from which the resource template is to be loaded. In general, the NULLHANDLE



```

MENU ID_RES
{
  SUBMENU "~File", MN_FILE
  {
    MENUITEM "~New\tCtrl+N", MN_NEW,,MIA_CHECKED
    MENUITEM "~Open...\tAlt+Ins", MN_OPEN
    MENUITEM "~Delete", MN_DELETE
    SUBMENU "~Save", MN_SAVE
    {
      MENUITEM "Save ~As...", MN_SAVEAS,,MIA_DISABLED
      MENUITEM "Save A~ll", MN_SAVEALL
    }
    MENUITEM "P~lace", MN_PLACE
    SUBMENU "~Printer", MN_PRINTER
    {
      MENUITEM "Pa~ge setup...", MN_PAGESETUP
      MENUITEM "P~rinter setup...", MN_PRINTERSETUP
    }
    MENUITEM SEPARATOR
    MENUITEM "E~xit", MN_EXIT
  }
  MENUITEM "~Window", MN_WINDOW
  MENUITEM "~Help", MN_HELP
}
-----
ACCELTABLE ID_RES
{
  "N", MN_NEW, CTRL
  VK_INSERT, MN_OPEN, ALT
}
-----
#defineMN_MENU90
#define MN_FILE 500
#define MN_NEW501
#define MN_OPEN 502
#define MN_DELETE 503
#define MN_SAVE 504
#define MN_SAVEAS 505
#define MN_SAVEALL 506
#define MN_PLACE 507
#define MN_PRINTER 508
#define MN_PAGESETUP 509
#define MN_PRINTERSETUP 510
#define MN_EXIT 512
#define MN_WINDOW 540
#define MN_HELP 580

```

**Figure 9.16** The contents of the files MNU, HHH, and ACC generated by Menu Maker.

value is used to indicate that the menu template resides in the executable from which the *WinLoadMenu()* function is called. However, nothing prohibits that the module be different from the one currently executing. To get the handle of a module you must employ the function *DosLoadModule()*:

```
#define INCL_DOSMODULEGR
APIRET APIENTRY DosLoadModule( PSZ pszName,
                                ULONG cbName,
                                PSZ pszModname,
                                PHMODULE phmod);
```

<i>Parameter</i>	<i>Description</i>
pszName	Buffer containing the name of the object that might have caused a problem during the function's execution
cbName	Size of the buffer declared in the first parameter
pszModName	Name of the module
phmod	Address of an identifier of type HMODULE wherein the module's handle will be inserted
<i>Return Value</i>	<i>Description</i>
APIRET	Success or failure of the operation

The first two parameters identify, respectively, the buffer and its size as will be employed by the function *DosLoadModule()* in case of some execution error. With *pszModName* you indicate the name of the module of which you want to know the handle, which will then be stored in the fourth parameter. Generally, this function is used to access to some dynamic link library (DLL) containing one or more resources that will eventually be used in the application. This will be discussed in the next chapter, since it is not a viable solution at this stage. Furthermore, this solution would force the program's resource to be resident in a DLL, which, although possible, is a somewhat rare situation.

## *Parsing the MNU and HHH Files*

At this point, we have to code a *parser*. This is what we will do:

- Ask the user for the name of the project previously indicated when saving the menu structure
- Open the files with the extension MNU and HHH (the ACC file will not be parsed, since the information it contains does not affect the creation of a menu window, but simply the accelerator table, and therefore is not relevant in this context)
- Detect in the MNU file the first item, which can be either a MENUITEM or a SUBMENU
- Retrieve the text string of the item and the corresponding ID
- Issue the message MM\_INSERTITEM to the menu window, after appropriately filling in the MENUITEM structure with the data retrieved from the MMM and HHH files
- Repeat the last two steps until the end of the MNU file is reached

A great deal of the inner workings of the parser depends on your skill and knowledge in using pointers. There is a whole science about how to generate flexible and efficient parsers; but this is a subject we will not be concerned about. The proposed algorithm is just a straightforward solution among several that could be devised, and may not be the best one.

## *The MENUITEM Structure*

Each single item that appears in a window of the type `WC_MENU` is fully described by the data contained in a structure of the type `MENUITEM`. The aspects that characterize it are: its position, styles, attribute, ID, handle of the possible associated drop-down, and a bitmap (if present). This is always valid, whether the item is a `SUBMENU` or a `MENUITEM`.

```
#define INCL_WINMENUS
typedef struct _MENUITEM
{ // mi
    SHORT iPosition ;
    USHORT afStyle ;
    USHORT afAttribute ;
    USHORT id ;
    HWND hwndSubMenu ;
    ULONG hItem ;
} MENUITEM ;

typedef MENUITEM *PMENUITEM ;
```

When generating menus starting from the text contained in the `MNU` and `HHH` files, it is necessary to retrieve all of the data required by the `MENUITEM` structure. Notably, the `MENUITEM` structure does not contain the text of the menu item: this piece of data must be passed to the window by means of the message `MM_SETITEMTEXT`. Let's now examine, one by one, the members of the `MENUITEM` structure in order to determine the source of the data in the two files, `MNU` and `HHH`.

## *The Position*

It is very simple to determine the position where to insert each menu item described in the menu template; you only need to specify `MIT_END` in order to have the inserted element appended to the bottom of the list of the previous ones. This rule applies even to drop-down menus, first-level or second-level. The position of the top-level menu is always counted starting from zero from the extreme left. This is computed by the parser when reading the `MNU` file in the search for the first level `SUBMENU` string.

## *The Style*

The styles of `WC_MENU` window items are defines having the prefix `MIS_`, as described in Table 6.5. In the case of Menu Maker, the only three styles supported are:

MIS\_TEXT—the default value—MIS\_SUBMENU, and MIS\_SEPARATOR. It is the parser's duty to define the appropriate style. Some considerations:

- The style MIS\_SEPARATOR appears only where there is a MENUITEM directive in the MNU file
- MIS\_SUBMENU is never present in the MNU file, but must be enforced by the parser whenever it encounters the string SUBMENU
- Also, MIS\_TEXT never appears in the template generated by Menu Maker, and must be declared for a generic menu item for a top-level, but not with an MIS\_SEPARATOR

## *The Attributes*

The attributes of the menu item are selected among those described in Table 6.7. Actually, the only two that we will encounter in the menu template, except for hand-crafted changes, are: MIA\_DISABLED and MIA\_CHECKED. These correspond to two of the three check boxes that are present in the lower part of the modeless dialog.

## *The ID*

Each menu item must have an ID, as we know. During the parsing of the MNU file, this information is retrieved from the HHH file and automatically assigned to the appropriate member in the MENUITEM structure. As we will see, the problem is not computing the IDs, but rather assigning them when defining a template. The ID must always be an integer number in the range between 0 and 65,536. Notably, a SEPARATOR will be missing an ID when implemented like MENUITEM SEPARATOR. On the other hand, if it is defined through the style MIS\_SEPARATOR then it must be associated with an ID.

## *The Submenu Handle*

It is necessary to specify the handle of a window of the class WC\_SUBMENU only for those menu items of the menu bar or of a drop-down that in turn implies a drop-down. In any other case, it is sufficient to specify a NULL. This handle is an ordinary handle to a window created with the function *WinCreateWindow()*.

## *The Bitmap Handle*

As was illustrated in Chapter 6, the items of a menu can be bitmaps in addition to text. This possibility is not supported by Menu Maker, and therefore the `hItem` member of the MENUITEM structure is never used in the program. The four bytes in the `LONG hItem` might therefore be employed as best suits the application. More generally, this space is accessible as a memory area for containing a pointer, a handle, or any other information that could be equal to the reserved memory area of each item in a listbox.

## *Saving a Menu Template*

Pressing the Save... button will cause the menu template generated by the application to be saved in a file. During this stage it will be necessary to inspect the windows of the class WC\_MENU present in the program. The algorithm implemented here is recursive in nature, as it has to account for nested drop-downs. The operations involved require the use of specific functions for preparing the strings to be stored in the MENU, HHH, and ACC files. The action performed by this pool of routines directly affects the later operations of the parser, so that tabs, commas, and white space are positioned in the documents according to precise rules that will eventually help the parsing operations.

## *Adding a Top-Level*

Once the text typed in by the user has been retrieved, it has to be purged of all spurious characters, like tildes. Then it is inserted in the specific listbox present in the dialog, and finally selected.

```

...
// insert an item in the listbox
sPos = (SHORT)WinSendMessage( CTRL( hDlg, DL_LIST), LM_INSERTITEM,
                              MPFROMSHORT( LIT_END), MPFROM( buff) );

// select a top-level menu
WinSendMessage( CTRL( hDlg, DL_LIST), LM_SELECTITEM,
                MPFROMSHORT( sPos), MPFROMSHORT( TRUE) );
...

```

A rather subtle aspect in the creation of menus is that of the ID's assignment. The algorithm used here is very simple, and subject to many enhancements. The ID of the first top-level equals to the number 500, the second 540, the third 580, and so on. This means that the maximum number of menu items for each drop-down is 40. This quantity is defined inside the application, and can be varied if desired. The first menu item of the drop-down associated with the first top-level will have the ID of 501, with subsequent increments. Actually, it is not that important to assign specific IDs to the menu items inserted in the WC\_MENU windows. The application will not use these values in an imperative way: The algorithm implemented here will simply transfer to disk whatever was previously computed, but it would always be possible to redefine them with a plain counter based on the physical position of the menu item in the menu structure. Although top-level, the value of NULLHANDLE is specified for the mi.hwndSubmenu member of the MENUITEM structure. The modification of this value takes place only after the user defines a drop-down to be associated to that top-level.

```

...
sID = MN_START + sPos * TOTITEM ;

mi.iPosition = sPos ;
mi.afStyle = MIS_TEXT ;
mi.afAttribute = 0 ;

```

```

mi.id = sID ;
mi.hwndSubMenu = (HWND)NULLHANDLE ;
mi.hItem = (ULONG)NULLHANDLE ;

// add the top level to the menubar
WinSendMsg( menu, MM_INSERTITEM,
            MPFROMP(&mi), MPFROMP( buffer)) ;
( buffer) ;
...

```

## Adding a Drop-Down

It is much more difficult to insert a drop-down. In the first place, you must determine the name of the top-level selected in the listbox of the modeless dialog. This will indicate where the new item is to be inserted, and allow its ID to be computed.

```

...
// determine the selected top-level
sPos = (SHORT)WinSendMsg( CTRL( hdlg, DL_LIST), LM_QUERYSELECTION,
                        MPFROMSHORT( LIT_FIRST), 0L) ;
...

```

If it is a first level drop-down, you must check that the appropriate WC\_MENU class has been created. This information can be gotten by examining the hwndSubMenu member of the corresponding top-level.

```

...
if( !mi.hwndSubMenu)
{
    CHAR szString[ 40] ;
    ...
}

```

If there is no drop-down, you will have to create it through *WinCreateWindow()*: The parent window is `HWND_OBJECT`, while the owner window is the application's menu bar.

```

...
// create new WC_MENU window
mi.hwndSubMenu = WinCreateWindow( HWND_OBJECT, WC_MENU,
                                NULL, 0L,
                                0L, 0L, 0L, 0L,
                                hmenu, HWND_TOP,
                                0L,
                                NULL, NULL) ;
...

```

Once this operation has been performed, you must modify the style of the top-level in order to set the `MIS_SUBMENU` flag:

```

...
mi.afStyle = MIS_TEXT | MIS_SUBMENU ;
...

```

The next step is necessary in order to avoid a minor annoying bug of PM. To modify the attributes of the previously created top-level, it is enough to issue the message WM\_SETITEM. However, this operation will make the underscore disappear from the character defined as mnemonic code. To avoid this, you can retrieve the text of the top-level, including the tilde symbols, and then reinsert it after changing its attributes with MM\_SETITEM.

```

...
WinSendMessage( hmenu, MM_QUERYITEMTEXT,
                MPFROM2SHORT( MN_START + sPos*TOTITEM,
                              sizeof szString),
                MPFROMP( szString) );

// set handle of drop-down menu in the top level structure
WinSendMessage( hmenu, MM_SETITEM,
                MPFROM2SHORT( 0, FALSE), (MPARAM) &mi );

WinSendMessage( hmenu, MM_SETITEMTEXT,
                MPFROMSHORT( MN_START + sPos * TOTITEM),
                MPFROMP( szString) );
}
...

```

From this moment on, any subsequent insertion of new menu items must refer to the drop-down, rather than to the menu bar. Now it is necessary to define the members of the MENUITEM structure in order to insert the new option in the menu window. The process is identical to that employed for the top-level.

## *Adding a SEPARATOR*

Of the four actions supported by Menu Maker, this one is the simplest. When filling in the MENUITEM structure, you can specify the style MIS\_SEPARATOR, but no attribute. The code will make appropriate checks to avoid the insertion of a separator bar in a top-level or as the first item of a drop-down.

## *Adding a Submenu*

To define a second level drop-down, you must first create a new window of the class WC\_MENU, according to the model described earlier. The operation must be performed twice if the top-level referenced does not have a drop-down of its own. Only a subsequent selection of the Submenu radio button will allow the interruption of the insertion sequence regarding a second level drop-down.

## Gathering Menu Maker Functions

In Menu Maker, you can tell subportions of the code apart so as to allow segmentation of the executable. The application's entrypoint is naturally the *main()* function, which does not call directly any function except for the application's window procedure through *WinDispatchMsg()*. Figure 9.17 shows the correlation that exists between the functions written in the application.

In the program there are three large accessory areas: the loading of a template, the saving of a template, and the editing of a template. Let's examine the portion saving a template produced by the user. The function *Save()* is the basis of the operation of saving in a file the template drawn on the screen. This function will in turn call the functions *CheckSubMenu()*, *StringParser()*, and *CalcMenuItem()*. The last two are even called by *MenuDlgProc()*, and this latter function is in turn called from *ClientWndProc()* alone. The four functions *CheckSubMenu()*, *StringParser()*, *CalcMenuItem()*, and *MenuDlgProc()* can thus be collected in a separate source file from that of the application. The function *Accel()* is called from *CheckSubMenu()* only if at least one of the menu items is associated with an accelerator. Since this will happen with a frequency that is difficult to assess beforehand (probably not very high), it is possible to set it apart in a segment of its own, and call the corresponding source file ACCEL.C. Analogous treatment can be reserved for *RemoveBlanks()*, putting it in a file called REMOVE.C.

The module definition file will then take on the following, more convoluted aspect:

Function	Called by
Accel	CheckSubMenu
CalcMenuItem	MenuDlgProc CalcMenuItem
CheckSubMenu	Save CheckSubMenu
ClientWndProc	main
MaskHandler	MenuDlgProc
MenuDlgProc	ClientWndProc
Project	ClientWndProc
RemoveBlanks	CheckSubMenu Accel
Save	ClientWndProc
StringParser	MenuDlgProc Save CheckSubMenu
stristr	Accel
strrep	CheckSubMenu
Sub1with2	CheckSubMenu

Figure 9.17 List of functions of the Menu Maker application.



```

..
CODE PRELOAD
DATA PRELOAD MULTIPLE

SEGMENTS
    SAVECODE32 LOADONCALL
    ACCELCODE32 LOADONCALL
    REMOVECODE32 LOADONCALL
...

```

Strongly related functions should be forced in the same code portion (page) by using the `alloc_text` pragma directive. If carefully planned, the paging activity dramatically decreases.



The relative complexity and length of the source code shown in Listing 9.1 requires a careful examination available only through reading the source code.

---

## Multithreaded OS/2 Applications

One of the features of the OS/2 operating system is its hardware-governed multitasking capabilities. The simultaneous execution of several applications is based on the partitioning of the CPU's time into very short intervals dedicated to the tasks waiting in line. The instantaneous switching between active tasks confers to the system all capabilities that are required for a true multitasking environment. The minimum interval that a task will persist in the CPU is equal to 32 milliseconds, while the maximum is by default approximately a quarter of a second. It is very unlikely that such a long interval will ever occur because of the competition among tasks.

The choice of the code to be processed is taken by the system's *scheduler* on the basis of priority; this is, in other words, true *preemptive* multitasking. In OS/2 there are as many as four distinct classes of priority—*time critical*, *fixed-high*, *regular*, and *idle* in decreasing order—each of which has 32 sub levels. By default, any process belongs to the *regular* class, but it can have its priority level increased or decreased through the function `DosSetPriority()`.

OS/2 has also introduced a new concept as far as multitasking is concerned on personal computers: the so-called *threads*. By this term one refers to the minimum quantity of code that can be addressed directly by the *scheduler* to the CPU. In general, a whole process coincides with the idea of a *thread*—the collection of several logically interrelated functions that are called to solve a specific problem.

### *Creating a Multithreaded Application*

To make the multitasking mechanism of OS/2 even more seamless, it is possible to design *multithreaded* applications consisting of several threads. In C Language a thread corresponds, in broad terms, to a function, including the contingent calls to other

functions present in the code. According to the specifications of the API of OS/2, a *thread* is a function that accepts a parameter of the type `ULONG`, and does not return any value, as illustrated in the following function prototype:

```
void EXPENTRY MyThread( ULONG u1Data) ;
```

Each thread possesses some distinctive characteristics: a stack of its own, an execution priority, and a set of system registers. The thread's stack is used for allocating identifiers with *block scope* and for passing parameters to the called functions. The execution priority is by default the same as the process to which the thread belongs, but it can easily be changed.

When a function of the code gets promoted to a *thread*, it will operate autonomously as far as CPU access is concerned, and will compete with all other processes/threads in execution, even with the one that spawned it. The *scheduler* will in fact send to the CPU a thread at a time, rather than a process at a time.

Splitting a process's functionality into several threads will favor the overall performance of a program's execution. Imagine, for instance, an application that needs to go through a long print session. This operation is generally slow and complex to the point of blocking the whole process. Implementing this functionality in a distinct thread will avoid this situation. The process can continue to operate while the print thread will perform the output operations.

## Creating a Thread

The first question to overcome when writing a multithreaded application is setting apart the portions of the application that can be taken out and profit from their own autonomy. In addition to the aforementioned example of a print routine, even loading a file from disk is an operation that could be accommodated in a distinct thread of the application, as is the recalculation of a spreadsheet, or the reading of data from the serial port.

The next step in developing a multithreaded application is defining the rules coordinating the activities of the single threads of the process in order to avoid anomalous and undesired behavior. The preferred tool for governing the activities of concurrent threads is usually represented by *semaphores* (muxwait or mutex, private or shared), which are very fast and reliable. Once this design phase has been settled, you can go on with writing the code.

We have already mentioned the overall structure of a function that will be promoted to a distinct thread. To transform a function into a thread, you have to call the function `DosCreateThread()`:

```
#define INCL_DOSPROCESS
APIRET APIENTRY DosCreateThread( PTID ptid,
                                PFNTHREAD pfn,
                                ULONG param,
                                ULONG flag,
                                ULONG cbStack) ;
```

<i>Parameter</i>	<i>Description</i>
ptid	Address of an identifier of the type TID
pfn	Pointer to the function that will take on the role of an independent thread
param	Parameter passed to the thread when it will be activated
flag	Initialization mode of the new thread
cbStack	Size of the thread's stack
<i>Return Value</i>	<i>Description</i>
APIRET	Success or failure of the operation

This function, like all *Dos* functions, will return the value of zero in the case of success or a positive value if an error takes place. The first parameter is the address of an identifier of type TID wherein the function will write the identification number of the thread. Each process in OS/2 holds a *process identification number* (PID) which is generated automatically by the system. This number will be in the range from 0 to 4 billion. Furthermore, each thread is characterized by a *thread identification number* (TID). For the primary thread, the one containing the *main()* function, this value is always 1, as we have already seen on many occasions by using SNOOPER. For all subsequent threads, the number is sequentially incremented, although always generated by the system. The secondary thread of a process is always spawned from the primary one; for the subsequent ones there is no fixed rule.

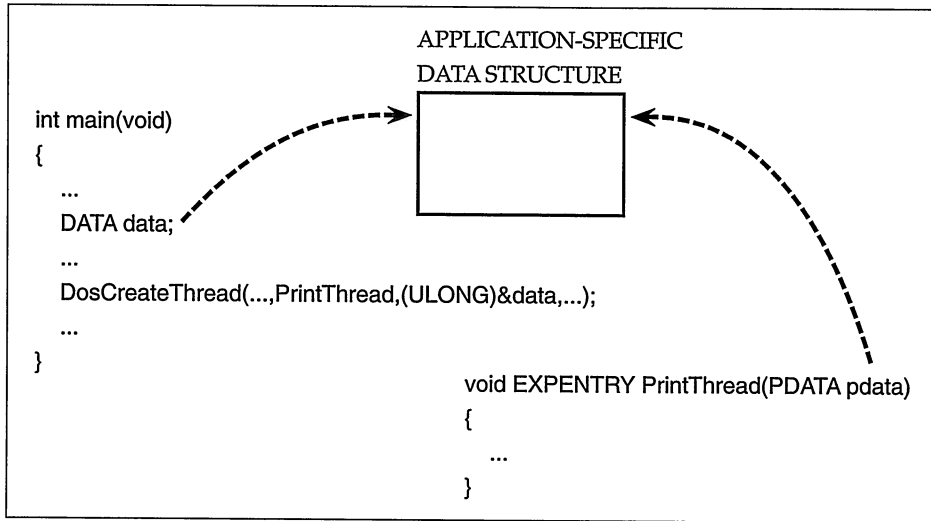
The size of a thread's stack is the last parameter in the syntax of *DosCreateThread()*. Typically, you will request a block of memory large enough to satisfy the overall requirements of the thread. There is no given algorithm for computing the size of a thread's stack; in general, this area is equal to 8KB or larger. The documentation of OS/2 lets us know that before calling any API function it is necessary to have a free stack space of at least 4KB. A thread's stack size of 8KB will therefore allow us to declare identifiers with block scope of 4KB.

A thread created with *DosCreateThread()* starts its execution as soon as the function returns based on the value of the *param* flag. The structure of the function that performs as a thread will allow it to exchange data with the generating thread. Therefore, it is not necessary to define identifiers with *source file scope* in order to communicate with the new thread. Often, the only parameter of a thread will act as a pointer to a data structure previously defined in the application, so that all communication needs between the existing thread and the one being spawned can be satisfied (Figure 9.18).

Returning to the previous example of printing, a specific thread could look like this:

```
void EXPENTRY PrintThread( PPRINTINFO pprnInfo)
{
    ...
}
```

where PPRINTINFO is some specific data structure implemented by the application. In the function's body it is possible to include any kind of code, even calls to other functions or the activation of other threads.



**Figure 9.18** Scheme of information passing between a thread being activated.

Among the run-time library functions of the C Set++ compiler, there is a special function called `_beginthread()`. In version 1.x of OS/2, the syntax of `DosCreateThread()` was different from the current one, and therefore the use of `_beginthread()` has its explanation. Both solutions are viable, although the API call is probably to be preferred over the runtime library function.

The handling of threads is completed by `DosSuspendThread()` and `DosResumeThread()`, which will suspend and resume the execution of a thread by specifying its TID. Both functions can be used exclusively within a process. The termination of a thread is performed with `DosKillThread()`, while the interruption of the current thread is achieved through `DosWaitThread()`.

## Compiling a Multithreaded Application

The compilation of a multithreaded application does not require any special provisions, as in the past with the selection of special libraries. Both with IBM's as well as with Borland's compiler, the whole process is simply that of selecting a specific radio button when creating a project.

## Threads and PM Applications

Up to this point we have written several PM applications that were all strictly single threaded. An OS/2 process is by definition made up of at least one single thread; i.e., the primary thread that features the TID of 1. The primary thread of a PM process is distinguished by the presence of the message queue, created by `WinCreateMsgQueue()`. Calling this function will cause a switch to the PM Screen Group. If you need to implement a multithreaded application, the first problem that arises is the message

queue. Is it necessary for each PM thread in a PM application to have a distinct message queue? The answer is yes and no at the same time: It really depends on the specific functional requirements designed into the application.

A PM thread characterized by a message queue of its own can perform all the activities that are typical of this screen group: It can receive and send messages and communicate with other windows. Without a message queue, most of these operations are not possible. If you need a thread capable of sending messages, then you must create a message queue and thus have an *anchor block*:

```
...
VOID EXPENTRY SecondThread( PINFO pInfo)
{
    HAB hab ;
    Hwnd hwnd ;
    HMq hmq ;

    hab = WinInitialize( 0 ) ;
    hmq = WinCreateMsgQueue( hab, 0L ) ;

    WinSendMsg( pInfo -> hwnd, WM_USER, 0L, 0L ) ;

    WinDestroyMsgQueue( hmq ) ;
    WinTerminate( hab ) ;

    DosExit( EXIT_THREAD, 0 ) ;
}
...
```

However, nothing prevents a specific operation of a PM application from being performed in threads lacking a message queue. Imagine, for instance, some demanding and time-consuming operations, like the earlier printing example. In a single threaded code, the process that will generate the information for the output device is treated as a single entity by the system's *scheduler*. The program is fundamentally tied up in the creation of a print metafile on the disk; any interaction between the user and the application is impossible. Generally this behavior is represented by the designer changing the cursor into the hourglass icon.

The splitting of the code into several threads will allow complex operations, like printing, without totally blocking any kind of interaction between the user and the application. Before proceeding further, let's examine in some greater detail the aspects regarding the priority classes and the 32 associated levels present in OS/2.

## The Priority Classes

Figure 9.19 depicts the priority classes in OS/2. The priority of a process is automatically inherited from its parent. The first application that is executed in the system is PMSHELL.EXE, which in turn allows the execution of new processes. PMSHELL.EXE is effectively the progenitor of all processes in OS/2.

Furthermore, a process can be structured into several threads, each one of which can have a priority of its own. To discover the priority of a PM process, let's make some changes to SNOOPER so that it can display some descriptive information

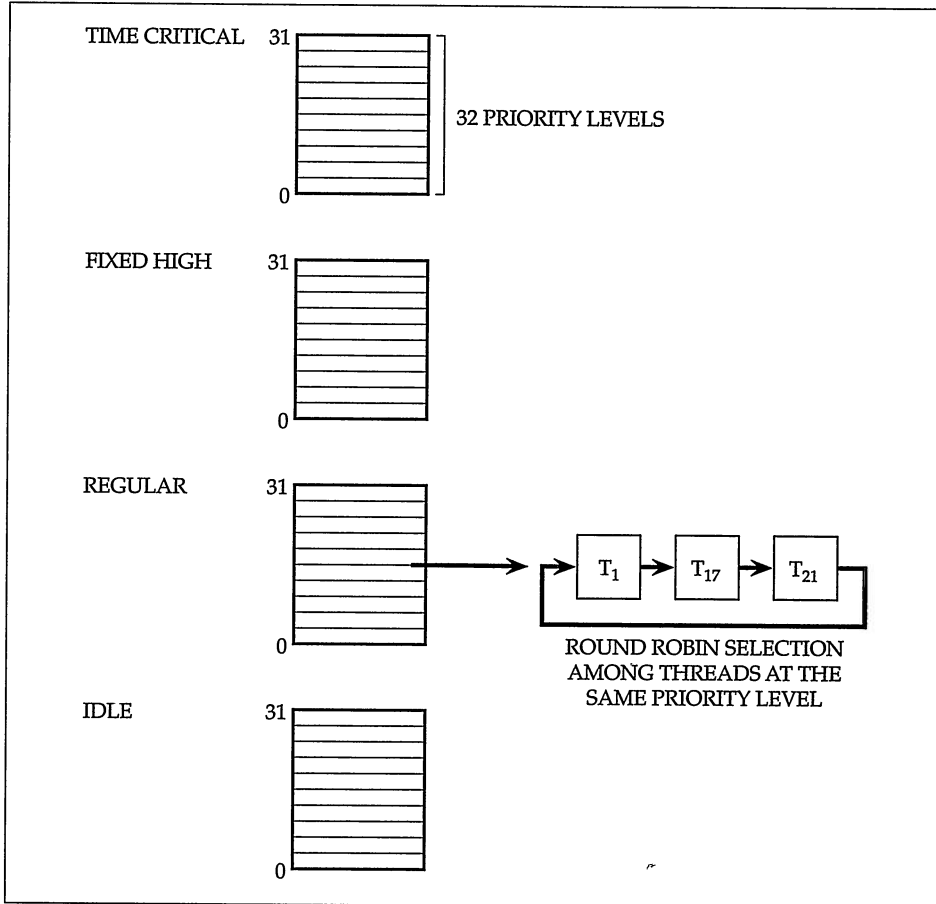


Figure 9.19 The classes of priority in OS/2 with their 32 sublevels.

regarding this. Let's examine in detail how *DosSetPriority()* works, in order to understand how to employ it in SNOOPER.

```
#define INCL_DOSPROCESS
APIRET APIENTRY DosSetPriority(  ULONG scope,
                                ULONG class,
                                LONG delta,
                                ULONG_PTR Tid) ;
```

Parameter	Description
scope	Object that is affected by the priority change: defines PRTYS_
class	New class: defines PRTYC_
delta	Change of priority
PorTid	PID or TID of the process or thread affected by the change of priority

<i>Return Value</i>	<i>Description</i>
APIRET	Success or failure of the operation

The first parameter defines which level must be obtained in the change of priority. The options available through the PM API are: PRTYS\_PROCESS, PRTYS\_PROCESSTREE, and PRTYS\_THREAD. These defines indicate, respectively, a single process, a process, and all its child processes generated with *DosExecPgm()*, a specific thread. The priority class, the second parameter, is one of the five defines present in OS2.H and summarized in Table 9.2.

PRTYC\_IDLETIME, PRTYC\_FOREGROUNDSERVER, PRTYC\_REGULAR, and PRTYC\_TIMECRITICAL refer to the four classes. With PRTYC\_NOCHANGE you can indicate that you don't intend to change the priority class, but only the priority level within the same class.

With the third parameter of *DosSetPriority()*, you establish the relative change of priority within the selected class. The numeric value of delta is a number between -31 and +31, a change relative to the current condition. There are also the constants PRTYD\_MINIMUM (-31) and PRTYD\_MAXIMUM (+31) to define a change to the maximum or minimum level.

The last parameter of *DosSetPriority()* corresponds to the number assigned by the system to the process (PID) or the thread (TID) that will be subject to the change in its priority. By specifying zero you can indicate the current process or thread. To know which is the current priority class, you can resort to *DosGetInfoBlock()*:

```
#define INCL_DOSPROCESS
APIRET APIENTRY DosGetInfoBlocks( PTIB ptib, PPIB ppib ) ;
```

<i>Parameter</i>	<i>Description</i>
ptib	Address of a structure of type TIB
ppib	Address of a structure of type PIB
<i>Return Value</i>	<i>Description</i>
APIRET	No value

Understanding how the *DosGetInfoBlocks()* function works is not straightforward. Its syntax requires the presence of a pointer to a TIB structure and a pointer to a PIB structure. Data about the current thread will be written in the first structure, while the second will receive data about the current process.

**Table 9.2** Flags for the *DosSetPriority()* Function

<i>Class</i>	<i>Value</i>	<i>Description</i>
PRTYC_NOCHANGE	0	No change of class
PRTYC_IDLETIME	1	Idle time class
PRTYC_REGULAR	2	Regular class
PRTYC_TIMECRITICAL	3	Time critical class
PRTYC_FOREGROUNDSERVER	4	Fixed-high class

```

struct tib_s
{
    PVOID tib_pexchain ;
    PVOID tib_pstack ;
    PVOID tib_pstacklimit ;
    PTIB2 tib_ptib2 ;
    ULONG tib_version ;
    ULONG tib_ordinal ;
} ;

typedef struct tib_s TIB ;
typedef struct tib_s *PTIB ;

```

Actually, the TIB structure contains within its member a pointer to a TIB2 structure that finally contains a member with the TID value:

```

struct tib2_s
{
    ULONG tib2_ultid ;
    ULONG tib2_ulpri ;
    ULONG tib2_version ;
    USHORT tib2_usMCCount ;
    USHORT tib2_fmCForceFlag ;
} ;

typedef struct tib2_s TIB2 ;
typedef struct tib2_s *PTIB2 ;

```

The acquisition of a thread's TID is thus the outcome of writing code like this:

```
ptib -> tib2_ptib2 -> tib2_ultid ;
```

The class and the priority level area are packed in the `tib2_ulpri` member—the class priority in the high-byte, the level in the low-byte. The `LOBYTE` and `HIBYTE` macros extract the two values:

```
ptib -> tib2_ptib2 -> tib2_ulpri ;
```

Unfortunately, `DosGetInfloBlock()` returns the information of the current thread (the one where the call is made). The previous `DosGetPrty()` function (no longer available) had one parameter to indicate the process ID, giving the opportunity to extend SNOOPER's capabilities to class and level priorities.

## Selecting a Priority Class

This first approach with priority classes immediately gives rise to some observations. In the first place, it is clear (from the action of SNOOPER) that all PM applications by default belong to the *regular* class with level of 0. The *scheduler* of OS/2 will always execute the thread that has the highest priority: In absolute terms, this is represented by the pair *time-critical*, level 31. The search for the next thread to be passed to the CPU takes place at preset time intervals, known as *time-slices*. In CONFIG.SYS you can



include the directive `TIMESLICE` to set the minimum endurance of a thread in the CPU; this value is at least equal to 31.25 milliseconds. The thread being executed when a time-slice expires will be taken off the processor if at that moment there is another thread in the system with a higher priority. Another directive that can be set in `CONFIG.SYS`, is `MAXWAIT`, which defines the maximum amount of time, expressed in seconds, that the CPU can wait upon a thread belonging to the regular class. When this time interval expires and a regular class thread cannot gain access to the CPU (it “starves” out of CPU cycles), its priority will temporarily be increased, so that it will be in a more favorable position the next time the scheduler selects a new thread to be executed. The temporary increase in priority (*boost*) can take place only in the regular class; therefore, it is more correct to speak about a variation in the priority level, and not in the class.

This behavior will suggest that you never change the priority of an OS/2 application, and accept the value provided by the parent process when it is activated. However, you can have a completely different approach to the single threads of a process. Often it is convenient to change the priority of a thread according to the requirements and possible interactions that happen within a process. The writing of multithreaded code is justified when by analyzing the code you can identify some operation that is a performance bottleneck. Changing the priority of a thread is a challenging exercise! To discover how to write a multithreaded application, let’s make some significant changes to `WHEREIS`.

---

## Multithreaded `WHEREIS`

The most complex and time-consuming task in `WHEREIS` is unquestionably searching for files in the selected drive. The routine implemented will scan through all directories in the current drive in the search for any file matching whatever was specified by the user in the specific entryfield. It is therefore the *search()* routine that is the natural candidate for being transformed into a separate thread within the process. A thread is, by definition, a function that does not return any value and that accepts a generic 32-bit parameter. In the specific case of `WHEREIS`, it is necessary to pass to *search()* the handle of the application’s window, a counter, and the text string corresponding to the files to be searched for. All these pieces of information will be gathered in a `DATA` structure declared in the application’s header file, `WHEREIS.H`.

```
// WHEREIS.H
...
typedef struct _DATA
{
    HWND hwnd ;
    CHAR szString[ 100] ;
    USHORT usTot ;
} DATA ;

typedef DATA * PDATA ;
...
```

The definition of the pointer PDATA, pointing to the data structure, allows you to implement the following strategy for passing all information from the primary thread to the secondary one:

- Declare an identifier of the type DATA with static storage class:

```
static DATA Data ;
```

- Whenever the Search button is pressed, the file counter is set to zero; this is the usTot member in the DATA structure:

```
...
case ID_SEARCH:
    WinQueryWindowText( hwndEdit, sizeof( buffer), buffer) ;

    // skip if there is non filename
    if( !*buffer || !strcmpi( buffer, Data.szString))
        break ;

    // copy text on pData
    strcpy( Data.szString, buffer) ;

    // erase counter
    Data.usTot = 0 ;

    // erase Found
    WinSetWindowText( hwndFound, "" ) ;
    WinEnableWindow( hwndDe1, FALSE) ;
    WinEnableWindow( hwndDe1All, FALSE) ;

    // erase listbox content
    WinSendMessage( hwndDir, LM_DELETEALL, 0L, 0L) ;
...

```

- Execute the search thread by calling *DosCreateThread()* and passing the address of the DATA structure to the new thread; from this moment on, the search thread is executing in parallel with the main application:

```
...
// activate the search thread
rc = DosCreateThread( &tid,
                    (PFNTHREAD)search,
                    (ULONG)&Data,
                    0L,
                    THREAD_STACK) ;
...

```

- Disable access to the Search button and to the listbox of drives before terminating handling of the message pertaining to the user's pressing the Search button:

```
...
// disable the search pushbutton
WinEnableWindow( hwndSearch, FALSE) ;

// disable the drive listbox
WinEnableWindow( hwndDrive, FALSE) ;
break ;
...

```

Let's examine how the search thread works. Although the operation of scanning through the disk directories is completely unconstrained by PM, it is nonetheless a PM thread. This explains the presence of the function *WinCreateMsgQueue()*. The choice of a second PM thread is ordained by the desire of filling in the listbox of files at the very moment they are found. The interaction with the listbox depends on issuing the message LM\_INSERTITEM by means of the *WinSendMsg()* function: In order to be able to call *WinSendMsg()*, a thread must necessarily have a message queue. Posting, via *WinPostMsg()* can be performed in a non-PM thread, since that operation does not imply a reply in the traditional means of communication through the message flow (*WinPostMsg()* will simply post a message and notify the success or failure of the call with a simple Boolean).

```

...
void EXPENTRY search( PDATA pData)
{
    ULONG ulDrv ;
    USHORT usDrv ;
    HAB hab ;
    HMQ hmq ;

    hab = WinInitialize( 0 ) ;
    hmq = WinCreateMsgQueue( hab, 0L ) ;

    DosQueryCurrentDisk( &ulDrvNum, &ulDrv ) ;
    schdir( pData, "\\\" ) ;
    DosBeep( 175, 300 ) ;
    ...
}
...

```

To avoid the presence of the message, it would have been necessary to design the application differently. Here's a suggestion. The purpose of the message queue is only that of supplying the listbox with information whenever a file name satisfying the search criteria is found. Instead of using a PM tool like *WinSendMsg()*, the secondary thread could fill in some memory area shared between the two threads. Since both threads belong to the same process, it wouldn't be necessary to resort to IPC tools like the allocation of a shared memory block through *DosAllocSharedMem()*. A simple, large array of char with *source file scope* would be more than adequate for this purpose. Disregarding this low-level solution, there is an abundance of others. An interesting one involves the pointer that was passed when the thread was activated. The DATA structure could be associated with a block of memory wherein you could store the file names as they are found. The initial abundant allocation is a concrete example for delayed commitments, resorting even to *page guard* techniques for establishing when it would be necessary to commit a new page.

The whole search logic takes place in the functions *schdir()* and *schfile()*. The first one is called from *search()*, while *schfile()* takes care of displaying the file name in the appropriate listbox.

```

...
void schfile( PDATA pData)
{
    ...
    sprintf( buffer, "%c:\\%s%s", ulDrvNum + 'a' - 1,
             strlwr( dbuf), strlwr( fileFind.achName));
    WinSendMsg( CTRL( pData -> hwnd, ID_DIR), LM_INSERTITEM,
               MPFROMSHORT( LIT_END), MPFROMP( buffer));
    sprintf( buffer, "%u", ++pData -> usTot);
    WinSetWindowText( CTRL( pData -> hwnd, ID_FOUND), buffer);
    ...
}
...

```

After retrieving the full file *pathname*, its text is inserted in the listbox, in sequential order, by calling *WinSendMsg()*. The file counter is automatically updated with a following call to *WinSetWindowText()*. The final part of *search()* takes care of enabling interaction with the drive listbox, activating the Search button, and finally, terminating the thread correctly.

```

...
// enable the search button and the drive listbox
WinEnableWindow( CTRL( pData -> hwnd, ID_SEARCH), TRUE);
WinEnableWindow( CTRL( pData -> hwnd, ID_DRIVE), TRUE);

WinTerminate( hab);
WinDestroyMsgQueue( hmq);
...

```

## Some Considerations

One of the essential rules in programming PM is trying to handle *any* message arriving in a window procedure in less than 1/10 of a second. The single threaded WHEREIS, presented in Listing 8.2 of Chapter 8, did not follow this rule. The CPU remained locked during the processing of a message previously retrieved from the queue with *WinGetMsg()* or received directly in the window procedure. Often the search for matching files takes considerably more than 1/10 of a second, and thus violates the fundamental rule of the event-oriented multitasking of the primary threads of any OS/2 application. During the actual search, the user cannot perform any operation, except for waiting for the whole disk to be scanned: Naturally, the wait is in no way dependent on the appearance of the hourglass cursor, but rather on the impossibility of taking the currently executing thread away from the processor (Figure 9.20).

Even the pressing of the keyboard combination Ctrl+Esc will not produce any effect. Only after the whole scan has been performed will the Window List be displayed at the center of the screen (Figure 9.21).

On the other hand, the suggested multithreaded implementation will hold the WM\_COMMAND pertaining to pressing the Search button only for the time necessary for triggering the search thread. Even during this time, the application is ready to respond to the user's actions, like moving the window on the screen, or the activation of another

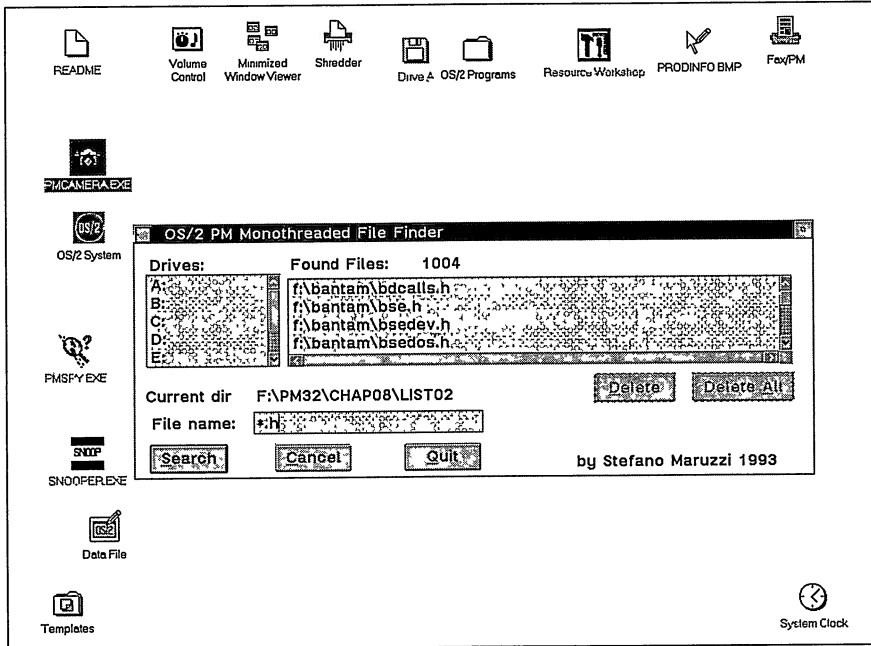


Figure 9.20 Execution of the single-threaded WHEREIS: The system is locked until the end of the search.

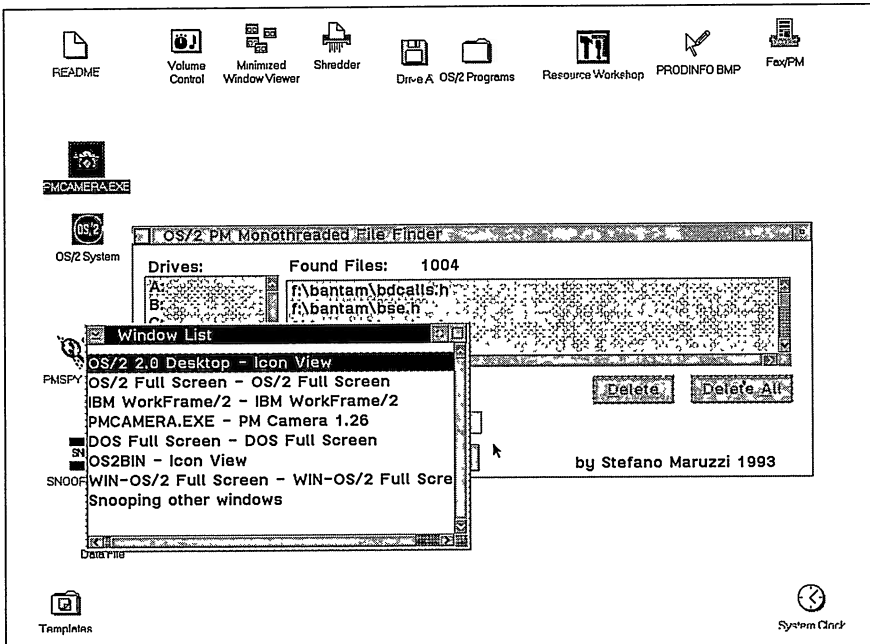


Figure 9.21 Window List is displayed in the PM screen group only after the whole file search has been terminated.

application, for instance. Figure 9.22 illustrates this behavior: Despite the fact that the search is not yet completed (as can be inferred from the number of matching files shown in the upper part of WHEREIS, immediately beneath the titlebar), it is possible for the user to move any window on the screen.

## Performance Tuning

There are many possible enhancements to make the execution of WHEREIS even faster. In the first place, the search thread could be promoted to the *time-critical* class in order to be more “appealing” to the scheduler whenever a time-slice expires. Furthermore, the logic followed in WHEREIS implies that the search thread be executed every time the user presses the Search button. The thread automatically dies at the end of the search. The call to *DosCreateThread()* is repeated several times in the program, and requires a certain amount of execution time.

A somewhat different design would involve the activation of the search thread as soon as the application is loaded, for instance when the message *WM\_CREATE* is received. The coordination between the search thread and the main process relies on a semaphore, controlled by both the primary as well as by the search thread. Here is the scheme to be implemented: The application starts and enables an *event* semaphore that will keep the search thread from executing. When the user presses the Search button, the primary thread will disable the semaphore and thus let the search thread execute. The entire code

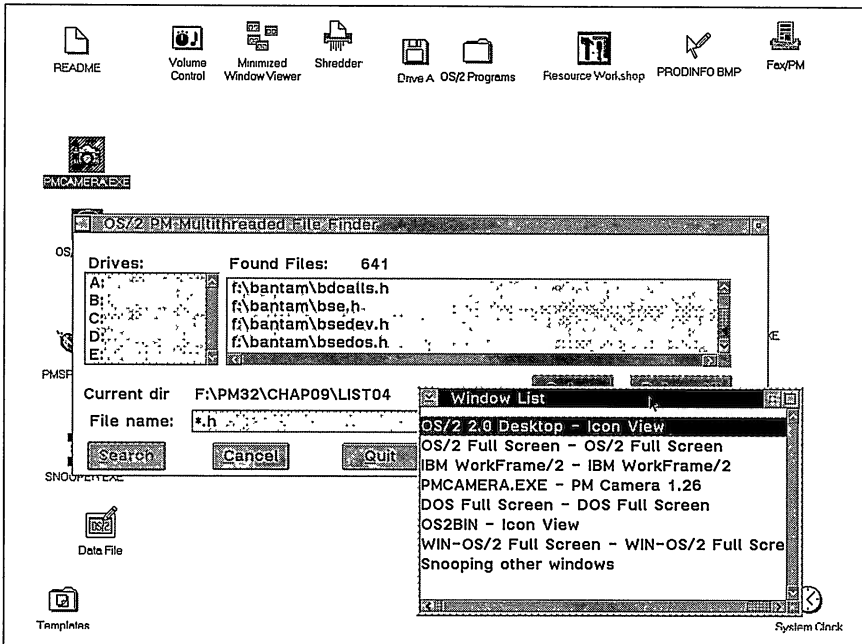


Figure 9.22 With the multithreaded WHEREIS, the user is not forced to wait (found files reached 641, compared to 1004 in Figure 9.21).

of the `search()` function is changed into an infinite loop in order to prevent the thread from destroying itself once the first search is over. The handling of the semaphore by the second thread is limited to calling the function `DosWaitEventSem()` immediately after the `while` statement: The only decision to be made pertains to the criteria to follow in displaying the pathnames found by the search, considering that the listbox is unique.

A second change involves the application logic. In the present implementation the search button is disabled while the secondary thread is running. Nothing prevents the application from starting many search-threads at the same time. The same piece of code can act as an independent thread in execution with no limitations at all. The bottleneck here is the output surface. In WHEREIS there is only one listbox to display the results. The obstacle is bypassed by creating a new window each time the user presses the search button or adopting a different interface (for example the matched file can be located in a different page in a multi-page notebook).

### Yet More Enhancements



The version of WHEREIS shown in Listing 9.2 is enhanced by yet another tool for making the utilities operations even more flexible. A double-click on the mouse button will execute the selected file, provided it has the extension .EXE. If, instead, it is a file with the extension C, DEF, RC, MAK, or H, then the system editor is invoked and the selected file is loaded into it. Figure 9.23 shows the interaction between WHEREIS and E.EXE.

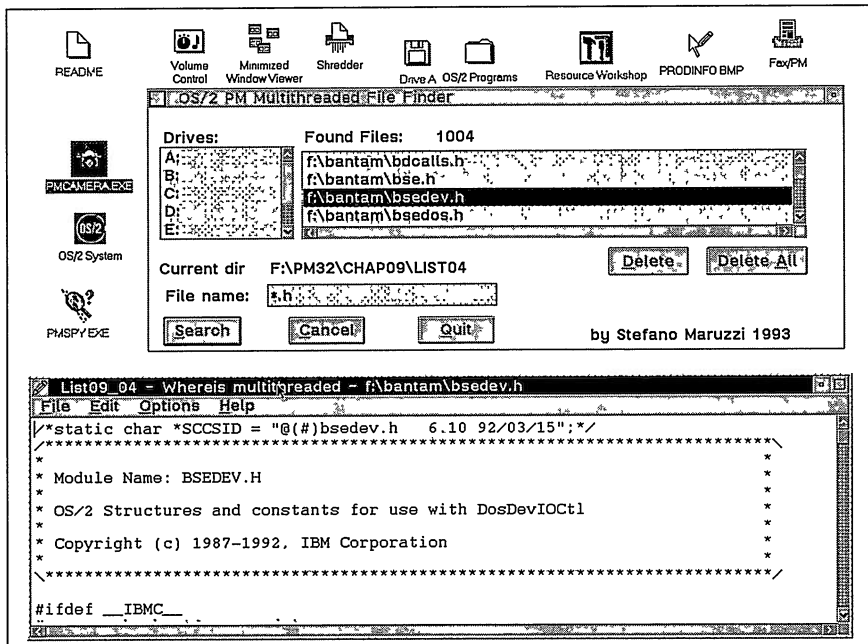


Figure 9.23 The system editor has been launched automatically after a double-click on the file with the extension of C.

The following code fragment refers to the portion of WHEREIS that handles this functionality. The function *DosExecPgm()* is called on both occasions with the flag of asynchronous execution, in order to give autonomous life to WHEREIS and the child processes generated in the meantime. As far as the system editor's execution is concerned, it is necessary to specify a text string corresponding to the command line of the system prompt. The syntax requires the presence of a null character to separate the arguments, and a double null at the end of the string.

```

...
case ID_DIR:
    switch( SHORT2FROMMP( mp1))
    {
        case LN_ENTER:
            {
                LONG i ;
                USHORT usItem ;
                CHAR *szExt[] = { ".c", ".def", ".rc", ".mak", ".h"} ;
                CHAR szPath[ 200] = "C:\\OS2\\e.exe\0 " ;

                sPos = WinSendMsg( hwndDir, LM_QUERYSELECTION,
                                MPFROMSHORT( LIT_FIRST),
                                MPFROMLONG( 0L)) ;
                usItem = WinSendMsg( hwndDir, LM_QUERYITEMTEXT,
                                MPFROM2SHORT( sPos, sizeof( buffer)),
                                MPFROMP( buffer)) ;

                // execute it if it is an EXE file
                if( !stricmp( ( buffer + ( usItem - 3)), "EXE"))
                {
                    DosExecPgm( szString, sizeof( szString),
                                EXEC_ASYNC,
                                NULL, NULL,
                                &rc, buffer) ;
                }

                // start the system editor for an H, C, MAK, RC, DEF file
                for( i = 0; i < 5; i+ +)
                {
                    if( !stricmp( ( buffer + ( usItem - strlen(szExt[ i]))),
                                szExt[ i]))
                    {
                        // prepare arg string
                        strcpy( szPath + strlen( szPath) + 2, buffer) ;
                        DosExecPgm( szString, sizeof( szString),
                                    EXEC_ASYNC,
                                    szPath, NULL,

```



```
        &rc,  
        "e.exe") ;  
    }  
}  
break ;  
...
```

A second enhancement to improve performance deals with the deletion operation. This portion of code should be placed in a separate thread to speed up execution and to let the user interact with the system.



# Subclassing, Superclassing, and DLL

Messages are the principal means of governing OS/2's multitasking. Any physical input action performed by the user is transformed into a message that will eventually reach the *window procedure* of the class to which the target window belongs. Even the designer can resort to messages for executing a number of operations like changing the status of a menu item, activating a window, inserting a text string into a *listbox*, and others.

When the message is queued, the path it follows implies a more or less lengthy stay inside the application's queue and a later retrieval by means of *WinGetMsg()* within the appropriate loop. The choice of the target window procedure is performed through *WinDispatchMsg()*, on the basis of the window's handle, given in a QMSG structure. In Listing 10.1, MSGQUEUE, you can see the code of a simple application that monitors the contents of the message queue: The output of the program is shown in Figure 10.1.

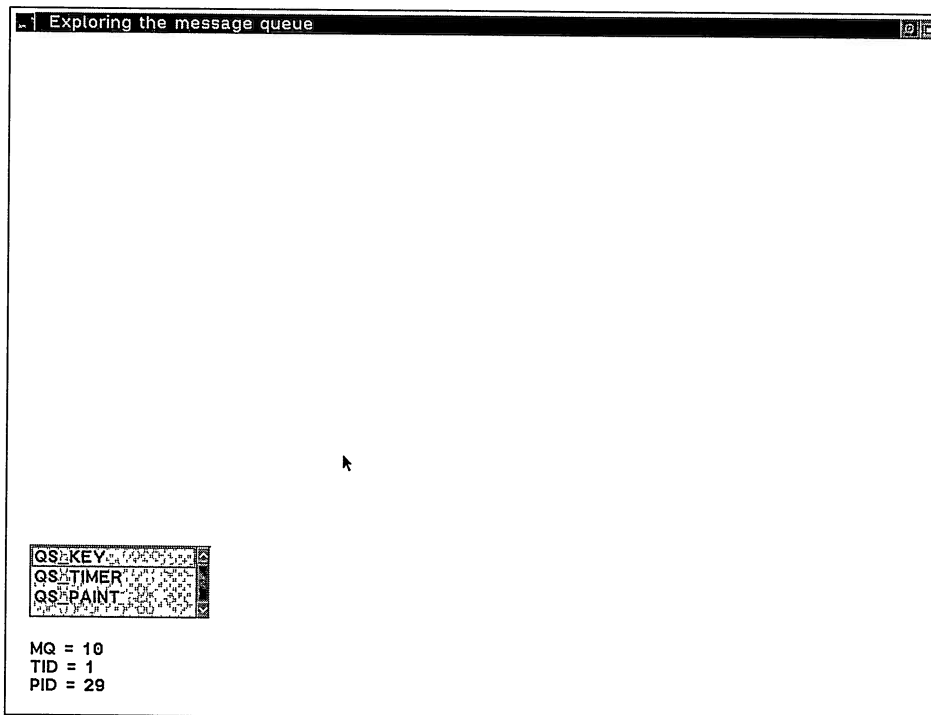


The functions used in MSGQUEUE are *WinQueryQueueInfo()* and *WinQueryQueueStatus()*. The first one allows you to examine, in general terms, the contents of a message queue.

```
#define INCL_WINMESSAGEMGR
BOOL APIENTRY WinQueryQueueInfo( HMQ hmq, PMQINFO pmqi, ULONG cbCopy );
```

<i>Parameter</i>	<i>Description</i>
hmq	Handle of the message queue
pmqi	Pointer to a structure of type MQINFO
cbCopy	Size of the MQINFO structure
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The MQINFO structure contains in the *msgs* member the overall number of messages currently present in the queue, in addition to giving the PID and the TID.



**Figure 10.1** PM does not provide any tool for examining in detail the contents of the message queue: You can find only the category to which the message belongs.

```
typedef struct _MQINFO
{ // mqi
    ULONG cb ;
    PID pid ;
    TID tid ;
    ULONG cmsgs ;
    PVOID pReserved ;
} MQINFO ;

typedef MQINFO *PMQINFO ;
```

With *WinQueryQueueStatus()* the application can get some information about the contents of the queue. The return value of this function corresponds to a define that summarizes the kinds of messages present in the queue.

```
#define INCL_WINMESSAGEGR
ULONG APIENTRY WinQueryQueueStatus( HWND hwndDesktop) ;
```

<i>Parameter</i>	<i>Description</i>
hwndDesktop	Handle of the desktop
<i>Return Value</i>	<i>Description</i>
ULONG	Set of QS_ defines used to indicate the contents of the queue.

A number of defines, as listed in Table 10.1, are packed inside this return value.

---

## Accessing the Window Procedure

The path followed by PM to get from a handle to the address of a window procedure is simple. This information appears in each window's reserved memory area. The function *WinQueryWindowPtr()*, together with the QWP\_PFNWP flag, will return the address of the window procedure associated with any specific window.

```

...
PFWNP pfnwp ;
...
pfnwp = (PFWNP)WinQueryWindowPtr( hwnd, QWP_PFNWP) ;
...

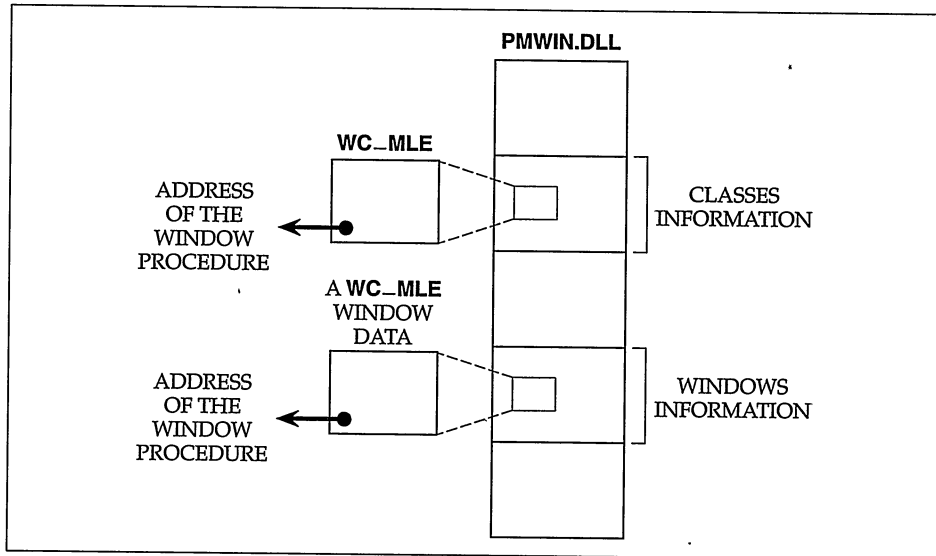
```

Once the window procedure's address is available, it is simple to call it with *WinDispatchMsg()*. You can do the same by calling *WinQueryClassInfo()*, as described in Chapter 7 (Figure 10.2).

**Table 10.1** The Flags Contained in the Value Returned by *WinQueryQueueStatus()*

<i>Flag</i>	<i>Value</i>	<i>Description</i>
QS_KEY	0x0001	There is a WM_CHAR in the queue.
QS_MOUSEBUTTON	0x0002	Event related to the pressing of a mouse button.
QS_MOUSEMOVE	0x0004	Event related to the movement of the mouse pointer.
QS_MOUSE	0x0006	Any kind of event associated with the mouse.
QS_TIMER	0x0008	There is a WM_TIMER message in the queue.
QS_PAINT	0x0010	There is a WM_PAINT message in the queue.
QS_POSTMSG	0x0020	There is a posted message.
QS_SEM1	0x0040	There is a WM_SEM1 message in the queue.
QS_SEM2	0x0080	There is a WM_SEM2 message in the queue.
QS_SEM3	0x0100	There is a WM_SEM3 message in the queue.
QS_SEM4	0x0200	There is a WM_SEM4 message in the queue.
QS_SENDRMSG	0x0400	There is a sent message.

---

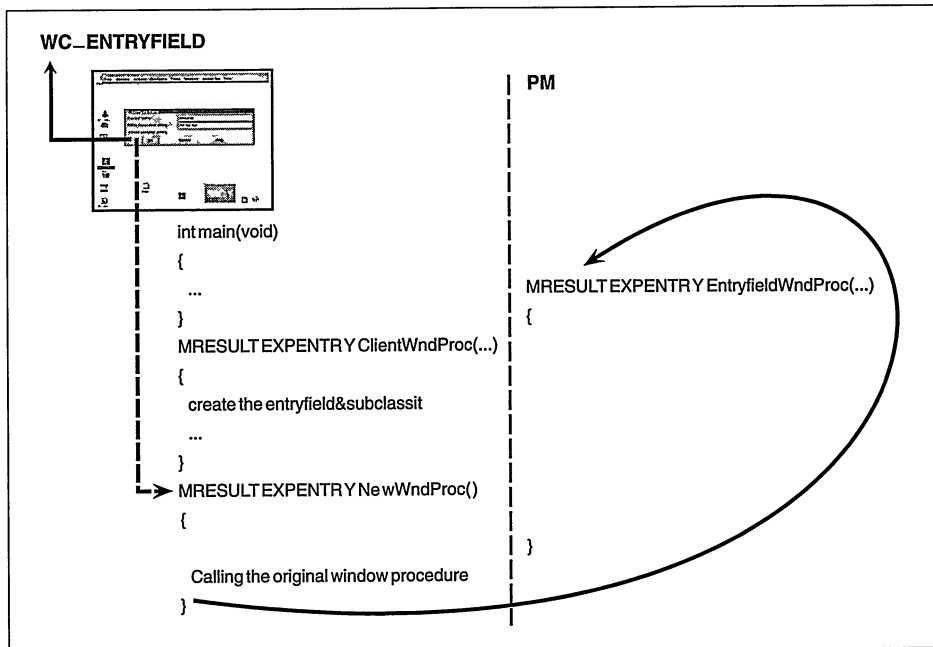


**Figure 10.2** Storage scheme of the window procedure's address in PM.

The presence of the window procedure's address in the reserved memory area of each window might appear to be redundant and purposeless, since, by definition, all windows belonging to the same class will have a common window procedure. However, the presence of a pointer to the window procedure associated with each window, in that window's reserved memory area, offers a significant advantage in writing PM applications.

The concrete outcome is that you can actually have *different* window procedures for each single window, despite their belonging to the same specific window class. This statement might seem to contradict what has been said in the previous chapters. Actually, nothing prevents the designer from registering different window classes if the plan is to use different window procedures, and thereby the designer would be able to discriminate the behavior of each window with respect to any event (a right mouse button click, for example).

The differentiation of window procedures makes sense only for those windows that belong to the predefined classes. In this case the designer is kept out of the information flow that takes place, for instance, between a *listbox* window and its window procedure, which is inside PM. The only possible method of interaction is to send specific messages available in the API (like the *LM\_* messages for the class *WC\_LISTBOX*), or to catch notification codes. But, if you were interested in knowing about the data generated by the system in response to any user-window interaction of a predefined class, the solution is *subclassing*. This term indicates a change in the reserved memory area of a window belonging to a predefined class: the change of the address of its own window procedure (Figure 10.3).



**Figure 10.3** Subclassing of a window belonging to a predefined class—in this case, an entryfield.

In order to perform subclassing, you must have two items: the handle of the involved window and the name of a new function that will play the role of new window procedure. To obtain the first item, you must create a window by calling *WinCreateWindow()* as this will be an instance of a predefined class. You must then write the new window procedure according to all rules governing ordinary window procedures. The only fundamental difference will be in the default processing reserved for messages coming to this procedure. In an ordinary window procedure, all messages—both those actually caught as well as those not catered for by specific case branches—will eventually reach *WinDefWindowProc()*, and thus be submitted to default processing. Instead, in the case of a subclassed window, it is much simpler to hand over the messages not pertaining to specific processing to the class's standard window procedure, which is defined internally in PM. It is thus possible to define *subclassing* as an operation consisting in temporarily deviating the message flow of a window belonging to a predefined class towards a window procedure managed by the designer, and then transferring information back to the original one for ordinary processing. Figure 10.4 illustrates this.

The scheme in Figure 10.4 is similar, in a certain way, to what has been seen in Chapter 8 regarding the role played by a *dialog procedure* in handling a *modal window*. For dialogs, this is on the whole, a natural behavior; instead, subclassing is enforced explicitly by the designer.

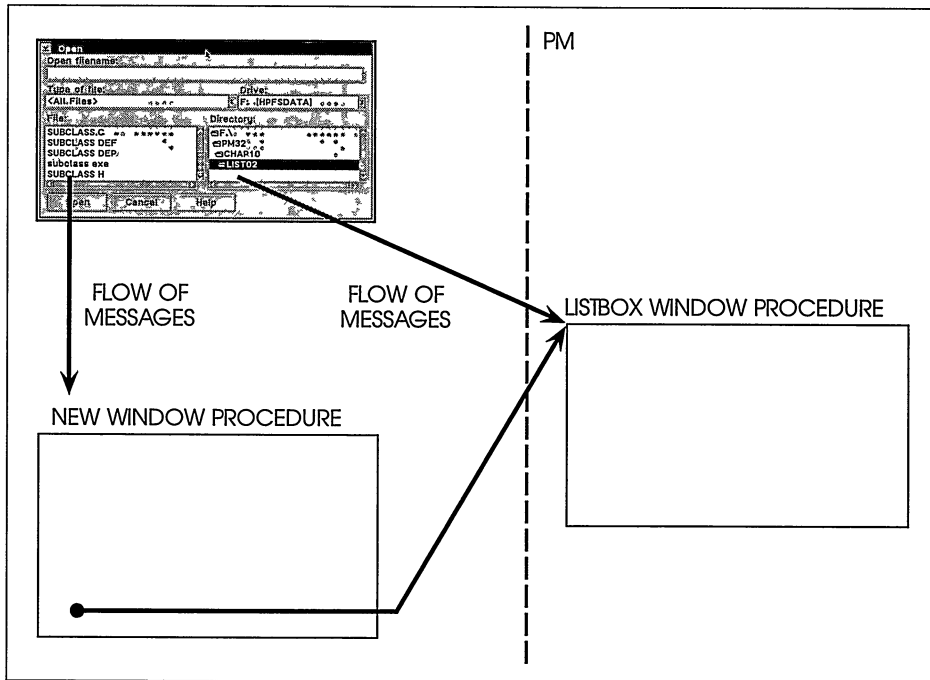


Figure 10.4 Implementation of subclassing for a window belonging to a predefined class.

## Performing Subclassing

There are two simple techniques that allow you to change the address of the window procedure associated with a window. The first alternative is calling the special function *WinSubclassWindow()*:

```
#define INCL_WINWINDOWMGR
PFNWP WinSubclassWindow( HWND hwnd, PFNWP pfnwp );
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window to subclass
pfnwp	Name of the function that will take on the role of new window procedure for the window

<i>Return Value</i>	<i>Description</i>
PFNWP	Address of the old window procedure

As it has been anticipated, the first parameter is the handle of the window to subclass, and the second one is the name of the new window procedure. The return value of this function is the address of the old window procedure. This piece of information is valuable because it is usually used to tell the new window procedure where it can find the original window procedure of the class.



Often, the identifier of the type `PFNWP` containing the return value of `WinSubclassWindow()` is a *source file scope* identifier. However, this approach is better avoided. A superior solution is sending a special message to the window immediately after it has been subclassed, and thereby transferring to the programmer defined window procedure the address of the original window procedure. The whole can be implemented in a code fragment like the following:

```
...
#define WM_PASSPROCWM_USER
...
PFNWP pfnwp ;
...
pfnwp = WinSubclassWindow( hwnd, NewProc) ;
WinSendMsg( hwnd, WM_PASSPROC, MPFROMP( pfnwp), 0L) ;
...
```

In the function `NewProc()` the message is caught in this way:

```
MRESULT EXPENTRY NewProc( HWND hwnd,
                           ULONG msg,
                           MPARAM mp1,
                           MPARAM mp2)
{
    static PFNWP pfnwp ;
    ...

    switch( msg)
    {
        case WM_PASSPROC:
            pfnwp = (PFNWP)mp1 ;
            return 0L ;

        ...
    }
    ...
    return (*pfnwp)( hwnd, msg, mp1, mp2) ;
}
```

The second solution, for performing subclassing of a window is calling `WinSetWindowPtr()`:

```
#define INCL_WINWINDOWMGR
BOOL WinSetWindowPtr( HWND hwnd, LONG index, PVOID p) ;
```

<i>Parameter</i>	<i>Description</i>
<code>hwnd</code>	Handle of the window to subclass
<code>index</code>	Index corresponding to the memory area to modify
<code>p</code>	Name of the function that will become the new window procedure for the window

<i>Return Value</i>	<i>Description</i>
<code>BOOL</code>	Success or failure of the operation

The flag to specify as the second parameter is always `QWP_PFNWP`, while the pointer is simply the address of the new window procedure. Since this function is not just for performing subclassing, it is necessary to precede this operation with the retrieval of the natural window procedure's address for the window, by calling `WinQueryWindowPtr()`, and then passing it to the appropriate function.

## A Sample Subclassing



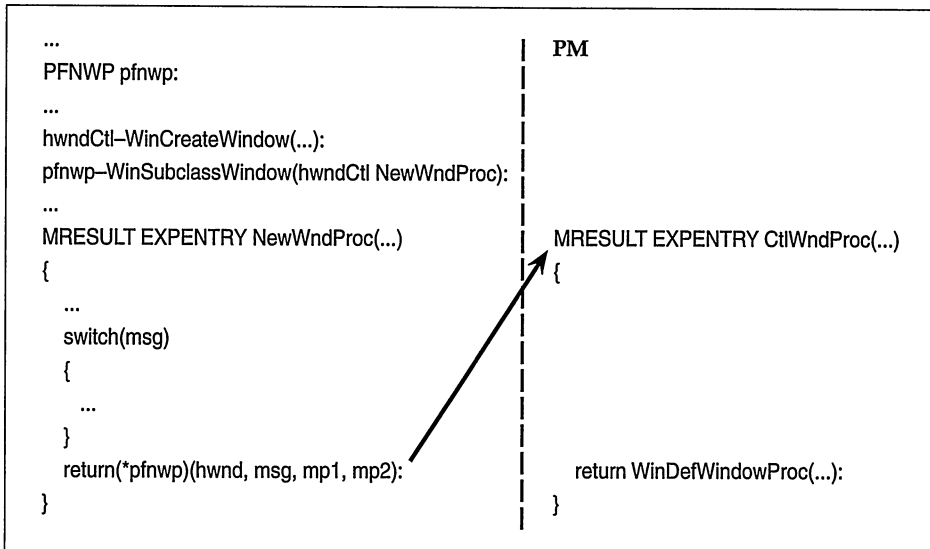
The code in Listing 10.2 shows a simple PM program that will create within the application client window two other windows belonging to the class `WC_MLE`, which share the available space of the client equally (Figure 10.5).

Of these two controls, the one on the left has been subclassed in order to catch the message `WM_CHAR` and prevent lowercase letters from being displayed and accepted by the window. Instead, when a lowercase letter is typed in, the routine will convert it into the corresponding uppercase, as shown in Figure 10.5.

The action is performed in the new window procedure of the *mle* on the left side of the application window, and is very simple and limited. However, what is important here is that the example shows that subclassing is an operation specific to each single window. As you can see, the second control will continue to process any kind of input, with no distinction whatsoever.

## When to Perform Subclassing

Subclassing a window is a very simple and straightforward operation—you only have to be careful to store the correct address of the original window procedure, and



**Figure 10.5** An application with two controls of class `WC_MLE`: The one on the left has been subclassed and will not accept lowercase letters.

remember to export the new function as an `EXPENTRY` in the module definition file (if present). Subclassing is thus an optimal solution when the application needs to change the typical functionality of a window of a predefined class only slightly. It is difficult to establish exactly how much message interception work should be performed in the new window procedure with respect to the original one. As a guideline, if the customization effort of the behavior of a window is limited to approximately 15–20 percent of its whole functionality, it is certainly better to resort to subclassing. Greater reworking could suggest the need for creating a new window altogether, starting with its registration.

Considering how frequently windows belonging to the predefined classes are used (as we have seen in Chapter 7), subclassing becomes a fundamental tool also for implementing behaviors like displaying a window context menu, or the handling of drag & drop operations.

---

## Superclassing

In the development of a PM business application, the presence of dialog windows should be consistent, both in terms of independent windows and notebook pages. Consequently, customized controls are often used. Assume you have to implement an accounting program. Windows of the class `WC_ENTRYFIELD` can be used as the primary source of user input in a number of situations: customer name, social security numbers, and other alphanumeric data.

Often, subclassing is an ideal solution. However, if you can identify in the code new kinds of windows used in several instances, subclassing each window becomes cumbersome. It is then much more practical to create a new class of window having most features in common with a predefined window class—like, for instance, `WC_ENTRYFIELD`. This approach to the problem is known as *superclassing*, which means generating a new class of window from the information available for some preexisting class, most often a predefined window class.

## Performing Superclassing

PM's API provides the function `WinQueryClassInfo()` in order to allow you to have in a `CLASSINFO` structure the data that was specified when a class of windows was registered. However, there is no provision that allows you to change these values directly. The reason for this is to protect the inner workings of the predefined classes. Just think what would happen if the system could change nature and behavior of the class `WC_SCROLLBAR`? All applications using a scrollbar would find themselves with an object with changed characteristics and probably would not work as expected.

The first step to perform is to retrieve information from a pre-existing window class. In Listing 10.3 this phase is performed directly after the interception of the message `WM_CREATE`:



```

...
case WM_CREATE:
    // retrieve information from the WC_ENTRYFIELD class
    if( !WinQueryClassInfo( HAB( hwnd),
                           WC_ENTRYFIELD,
                           &cls) )
        WinAlarm( HWND_DESKTOP, WA_ERROR) ;
...

```

In `cls`, an identifier of type `CLASSINFO`, you can find information pertaining to the name of the window procedure, the styles, and the size of the class's window words. In practice, these are the parameters of `WinRegisterClass()`. On the basis of these values, you can now register a new class. There are only a few values that need to be changed. Naturally, you must provide a new name—for example, `NewEdit`—and the address of the function that will take on the role of window procedure. The style and the size of window words, instead, are retrieved automatically from the appropriate members of the `cls` structure. The definition of the new window words must be added to the original outfit of the class. Furthermore, in order to avoid errors during the registration procedure, it is good to clear the style flag `CS_PUBLIC` in the `cls.f1ClassStyle` member. The style `CS_PUBLIC` is always set in the predefined windows; but it can be specified only if registration takes place inside a DLL. The following code fragment shows the correct way to register a new class of windows, in order to prepare for superclassing.

```

...
if( !WinRegisterClass( HAB( hwnd),
                      szClassName,
                      MyWndProc,
                      cls.f1ClassStyle & ~CS_PUBLIC,
                      cls.cbWindowData) )
    WinAlarm( HWND_DESKTOP, WA_ERROR) ;
...

```

The new class, `NewEdit`, relies on the services of the `MyWndProc()` window procedure and on those of the original class for processing all messages it receives. Before going on with the creation of the first window belonging to the new `NewEdit` class, you must tell the function `MyWndProc()` the address of the window procedure of the class `WC_ENTRYFIELD`. Even in this case you can resort to the mechanism we have seen earlier, by using the new message `WM_PASSPROC`. However, in respect to *subclassing*, it is not possible to call `WinSendMsg()`, because you do not have any handle. You must call the function `MyWndProc()` directly, and specify null handle and all remaining data pertaining to the message `WM_PASSPROC`.

```

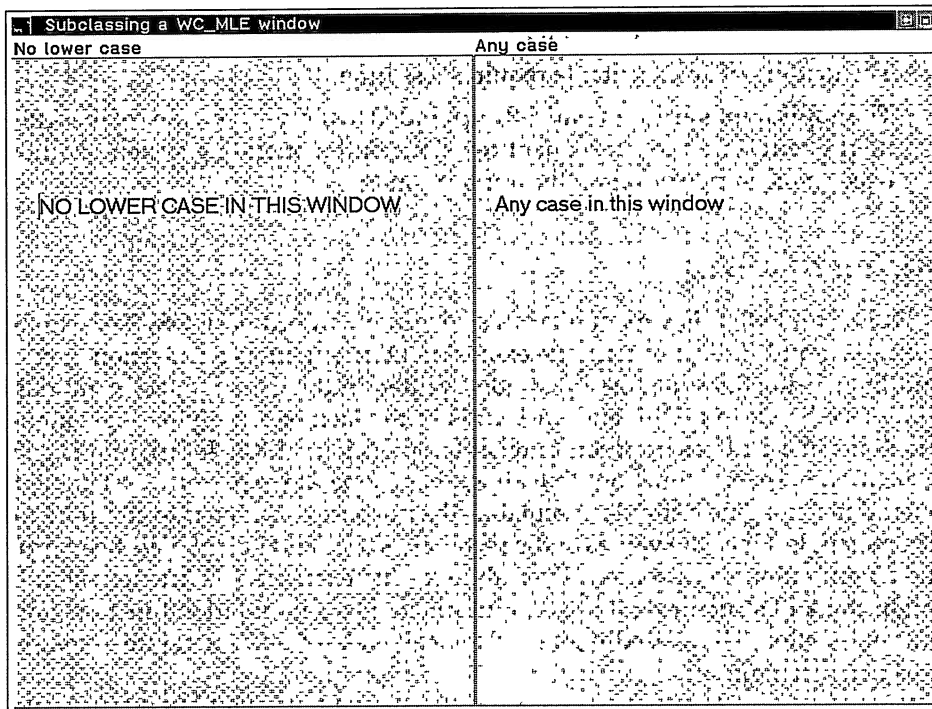
...
// pass the old wnd proc address to MyWndProc
(*MyWndProc)( NULLHANDLE,
              WM_PASSPROC,
              MPFROMLONG( (ULONG)cls.pfnWindowProc,
                          0L) );
...

```

Once you have registered the class and passed the address of the window procedure of the class from which the superclass has been derived, you can create the desired windows, and be certain that message flow will first reach the *MyWndProc()* function, then the internal PM function that is specific to the class *WC\_ENTRYFIELD*. In Figure 10.6 you can see a PM application that has two controls of the class *NewEdit*: Any lowercase input is converted into uppercase.

## Features of Superclassing

In PM, class registration is an operation that takes place for each single instance of a program. The scope of the class is therefore limited to the sole executable wherein it has been registered. This means that a multimodule application will have to register the same class of windows as often as appropriate. This situation is ideal in many development projects. On the other hand, if you want registration to take place only once in the system, and that creation of windows belonging to those classes be allowed both for executable modules as well as for DLL, then you have no other solution than to register the class directly inside a DLL. This DLL must then be loaded when you boot the system. This is essential to be certain that any process will be able to take advantage of the services provided by the new class.



**Figure 10.6** A sample PM application that uses controls generated by means of superclassing.

The terminology for this kind of operation is not *superclassing*, but creating customized controls. To do this, it is necessary to understand the design rules governing *Dynamic Linking Libraries*.

---

## Dynamic Linking Libraries

A frequent need in a multitasking environment is to reduce to the bare minimum the amount of memory occupied by all executing applications. In the development of OS/2, a number of solutions have been devised to limit the consumption of memory by the applications. In the previous chapter we explored the techniques used for producing segmented code.

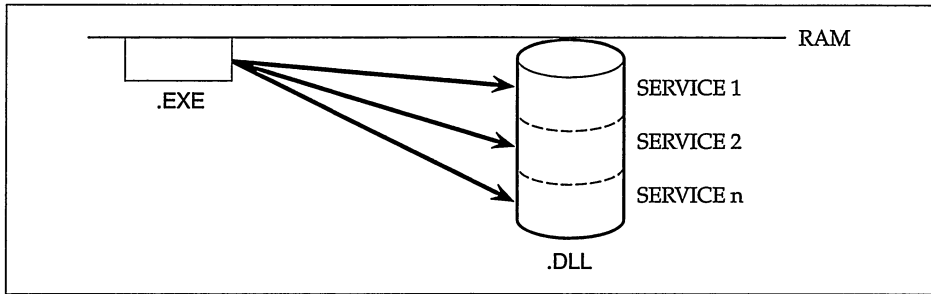
The fundamental strategy of OS/2 is to provide the software designer with a series of services—the API—which can be called directly from within any high level programming language. The API of OS/2 is actually accommodated in a number of modules known as *Dynamic Linking Libraries* (DLL). A typical PM application is characterized by the presence of a number of calls to the *Win* and *Gpi* services, in addition to *Dos*, *Prt*, *Drg*, and others. This means that a considerable portion of a PM application resides in one or more DLLs that are shared among several applications. There are many good reasons for this:

- The executable files are smaller
- The loading of an executable requires less time
- The applications are automatically upgraded to the new versions of the API which might be released in subsequent revisions of the operating system, without any need to recompile or relink the application

### *Definition of a DLL*

A DLL is a file that is very similar in structure to an ordinary OS/2 executable, for example, an EXE file, but with the extension DLL. A DLL is a collection of compiled and linked functions, almost invariably written in the C Language. A fundamental difference between a DLL and an EXE is in the lack of a unifying logic among the functions in a DLL, as opposed to what happens in an EXE. A program is generally structured in several functions that altogether cooperate to perform certain high level operations. Imagine a simple executable, the purpose of which is to compute the area of a rectangle. Once the values of the rectangle's base and height have been provided, the computation can proceed. Even though this example is extremely simple, you might think of structuring it in an input function, a computation function, and in an output function. The procedures are all logically related and cooperate to compute the area of a rectangle.

Now, let's examine the same program developed like a DLL. In this case, the DLL could effectively contain the three named functions, but there would be no unifying code taking advantage of these services. Some other executable will call the input



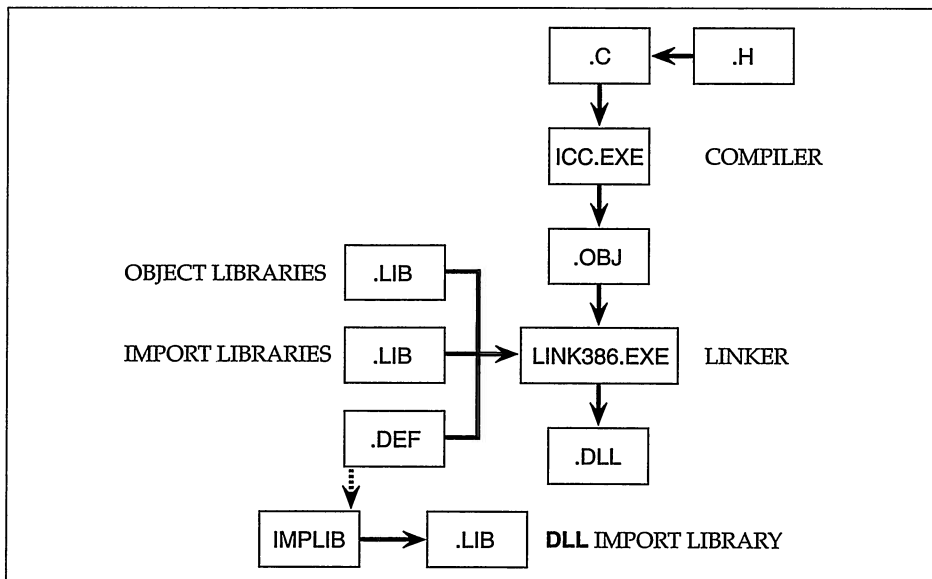
**Figure 10.7** Relationship between an executable and a DLL.

function, the computation function, and the output function in order to achieve its purpose. In practice, a DLL is a collection of services that can be called by other applications during their own runtime. Figure 10.7 illustrates the relationship between an EXE and a DLL.

### *How to Produce a DLL*

There are no different rules for producing a DLL with respect to what we have seen so far in the design of a PM application (Figure 10.8).

The starting point is always represented by a source code containing the functions that are intended to be called at a later stage by some other running application. A DLL is allowed to call other functions of OS/2's API, compiler runtime library



**Figure 10.8** Production scheme for an OS/2 PM DLL.

functions, or other functions within the same DLL. In all cases, it is necessary to include the appropriate header files. Once the code has been written, it can be compiled. You must indicate at the project level that you want to create a DLL rather than an EXE (the presence of the DLL extension is, in most cases, a purely cosmetic distinction).

Once you have obtained the object module, you can continue with the linking phase. LINK386 will produce an executable module deprived of the traditional entry point. As a matter of fact, the *main()* function will be missing altogether. This explains the different verbiage used for a DLL: An EXE is *executed*, while a DLL is *loaded* into memory.

In order to become fully operative, DLLs must physically be placed in some directory specified with the LIBPATH directive in CONFIG.SYS. If this is not the case, you will meet an error message informing you about the system's inability to execute the file \*\*\*, where the three asterisks sequence indicates one or more DLLs associated with the involved executable. The error message returned is 1804, and the explanation provided by the help system does not help much.

SYS1804: The system cannot find the file \*\*\*.

EXPLANATION: The file does not exist or is in a different directory.

ACTION: Check the spelling of the file name and its location.

Retry the command.

The distinction between a DLL and an EXE is not clearly manifest during the coding phase or for the nature of the file's contents (both files are compiled and linked modules). Rather, DLLs are to be considered portions of an executable that have been extracted from a program and parked in the system's memory in order to allow other modules to access to its functions or services.

There are not special limitations on the nature of the functions inside a DLL. Often the call convention `_System` (EXPENTRY) is used for consistency with the API, but even `_Optlink` will operate correctly. You need only be careful about listing in the DLL's DEF file the names of all functions that you want to be callable from outside the DLL (this operation is mandatory).

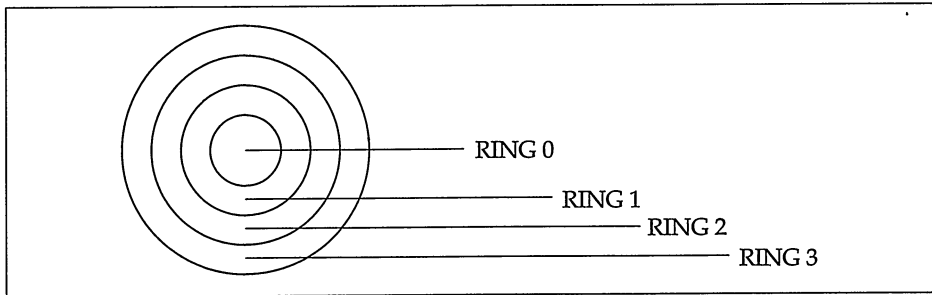
## Advantages of DLLs

DLLs are extremely important in OS/2. The operating system is structured in layers around the four rings of the 32-bit *protected mode* (Figure 10.9). The actual kernel and the device drivers operate in ring 0, the one with the highest priority. The applications operate in ring 3, the one with the lowest priority.

Of the four rings, the second one (number 1) is not used by OS/2. The layered nature of the whole system is shown in Figure 10.10. The DLLs containing the API of OS/2 are usually found in ring 3. This confirms the nature of "pseudo-applications" of DLLs. In some cases DLLs can be present in ring 2. This condition is necessary if you need to execute IOPL (*I/O privilege level*) code.

The role played by DLLs in making OS/2 work is fundamental. Actually, Presentation Manager is implemented as a series of DLLs that are loaded into memory



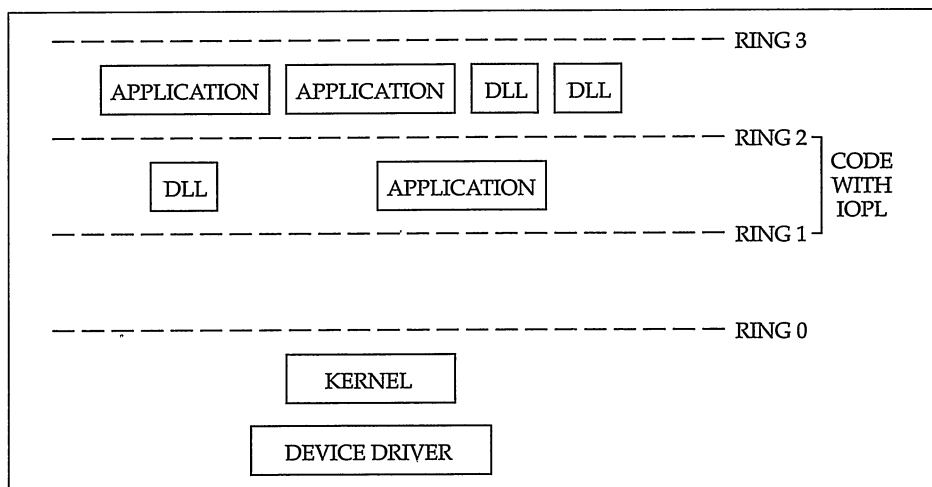


**Figure 10.9** The subdivision in layers of the 32-bit protected mode of the 386, 486, and Pentium in Intel's iAPX86 family of processors.

immediately after the initial bootup phase. Any PM application, starting with PMSHELL.EXE, will take advantage of the services present in PMWIN.DLL in order to act as windows on the computer's screen.

Nothing prevents the software designer from creating and adding new DLLs in addition to the standard outfit provided by OS/2. In principle, there is no difference between the DLLs provided by OS/2 and those developed in-house. All "home-brewed" DLLs will contain a collection of services that can be used by one or more applications. Knowledge about the capabilities of this new set of tools will of course be limited and not universal, as opposed to the case of OS/2's API, whose capabilities are known through the world of OS/2 developers, since it comes with the development kit's documentation.

A new kind of business has developed that markets professional DLLs containing useful functions (that enhance the system's API) that can ease the development and



**Figure 10.10** Layered architecture of the OS/2 operating system with respect to the processor's protected mode.

coding of OS/2 applications. Even more often, the creation of new, customized controls within DLLs offers interesting business opportunities for software houses, since they can be exploited even by users of macro languages, like the one in Lotus 1-2-3.

## Producing a DLL

Let's examine Listing 10.2, which shows how to create a simple PM application that can display a window on the screen. In the sample code you can find a number of calls to PM's API: *WinInitialize()*, *WinCreateMsgQueue()*, *WinRegisterClass()*, *WinCreateStdWindow()*, and many others. During the compilation phase, the C compiler cannot know anything about the nature of these functions, and therefore handles them as unknown objects. Consequently, it creates a table of external references, containing the name of the function, and, optionally, a description of its type. In performing these actions, the compiler is unaware of the nature of the calls: They might be calls to the API, calls to a DLL developed by the programmer, or even an ordinary C runtime library function call. As it does not have a complete picture of the situation (the compiler can act only upon one source file at a time), it cannot come to any conclusion and delegates to the linker the chore of solving all external references—computing the correct address of the function being called from within the code.

The linker will try to resolve any external reference in the following ways:

- It will examine all OBJ modules specified as the first argument in the linker's command line
- It will seek in all libraries (LIB) explicitly indicated in the linker's command line, or indicated automatically by the compiler
- It will read the area labeled by `IMPORTS`, if present, in the application's definition file

If none of these three phases produce information able to resolve the external reference, then the error message `L2029 unresolved external` will be displayed. Let's examine all possible situations.

## Static Linking

Static linking is what happens when an external reference is identified in one of the object modules specified during linking, or in a library like `DDExxxx.LIB`. This is typical of a segmented application, like Menu Maker (Chapter 9), where many functions reside in various modules that make up the program. Similarly, static linking takes place when any C runtime library function, like *sprintf()*, is used.

The linker will add to the original module (the first OBJ specified on the command line) the object module associated with the external reference, and thereby increase the size of the original module. Once this phase is over, it will always be the linker that resolves the problem of the unresolved addresses inside the program, by writing the physical address of the added function in the appropriate part of the call instruction (CALL). Figure 10.11 illustrates how static linking works.

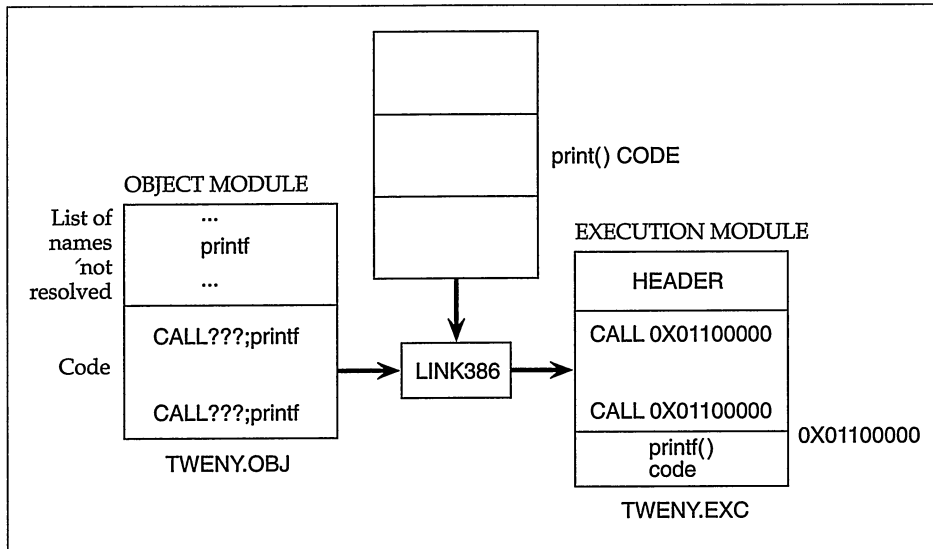


Figure 10.11 Scheme for resolving external references in static linking.

## Dynamic Linking

When the unresolved reference does not refer to some service contained in a DLL, the linker must act somewhat differently. However, it is obvious that the linker must fill in a "hole" in the program, the place where the address of the function to be called is missing. In the case of dynamic linking, the linker will not add the object module corresponding to the call, and the "hole" is filled in differently.

Let's examine what happens when linking a generic PM application that often resorts to calling API services. The fourth parameter of the linker is used to indicate the library OS2386.LIB which contains all references to the API of OS/2.

```
link386 machine,,, os2386, machine.def
```

Naturally, OS2386.LIB does not contain the code of all functions provided by OS/2 (several hundreds), but only references to the API. As a matter of fact, OS2386.LIB is a library used for generating a *relocation record* or even an *import library*. The concept of *import library* is strictly related to what happens in a DEF file, corresponding to the IMPORTS directive. By definition, all functions of the type EXPENTRY which are written by a programmer in a PM application must correspond to the EXPORTS directive. This allows the linker to compute the correct address of this code portions, which are later called by PM. The API of OS/2 is a set of services of the type EXPENTRY, and therefore it is reasonable to suggest that when the DLLs for OS/2 were written (like PMWIN.DLL), the EXPORTS area of the appropriate DEF file contained a list of all EXPENTRY functions declared in the code. In practice, when writing a DEF file for a DLL you must list all services that other applications (mainly EXE modules) will be

able to call from within their own code. The OS2386.LIB library is a typical example of an *import library*: a complete list of the API names of OS/2.

When the linker recognizes the table of external references produced by the compiler for the *Win* functions, it will find in OS2386.LIB the correct reference, and will insert that into the executable. You can think about the contents of OS2386.LIB, and generally of any import library, as a Rolodex: You might use it to collect business cards that you receive during meetings, business lunches, or other encounters during exhibitions and shows. Each business card will list information that allows you to contact some individual, by listing all items that are important for that purpose: first and last names, title, position, working address, phone number, fax number, and so on. The linker will insert into the executable the data about some service contained in an *import library*, and avoid error messages announcing unresolved externals. This does not mean that the executable is ready to be run. Only when it is loaded will it be possible to solve the problem and discover where in memory any specific function is located and thus obtain its address.

In practice, the executable that contains calls to services provided by DLL will not include the external service; it is enough to know the generic reference represented by the name of the DLL and the name of the service. In the case of the `IMPORTS` directive in the DEF file, writing a reference to an external service takes on the following form:

```
...
IMPORTS
    MIE01.MieEmphasizeWindow
    MIE01.MieCenterDlg
...
```

The information that has to be given to the linker is: the name of the DLL and the name of the service.

## *Structure of a Relocation Record*

The information contained in an import library or derived by the linker on the basis of what is specified in the `IMPORTS` section of a DEF file will allow the creation of a *relocation record*. The initial data falls into one of the following three categories:

- The external name, that is, the physical name of the function as it is encoded in the application code and as published by the compiler in the table of external references.
- The name of the DLL physically containing the code and the data of the function.
- The entry point of the function in the DLL. This piece of information is provided as a simple number or as the complete name.

The linker will define the relocation record in the application only when it has all the necessary information, whatever the source of it might be:

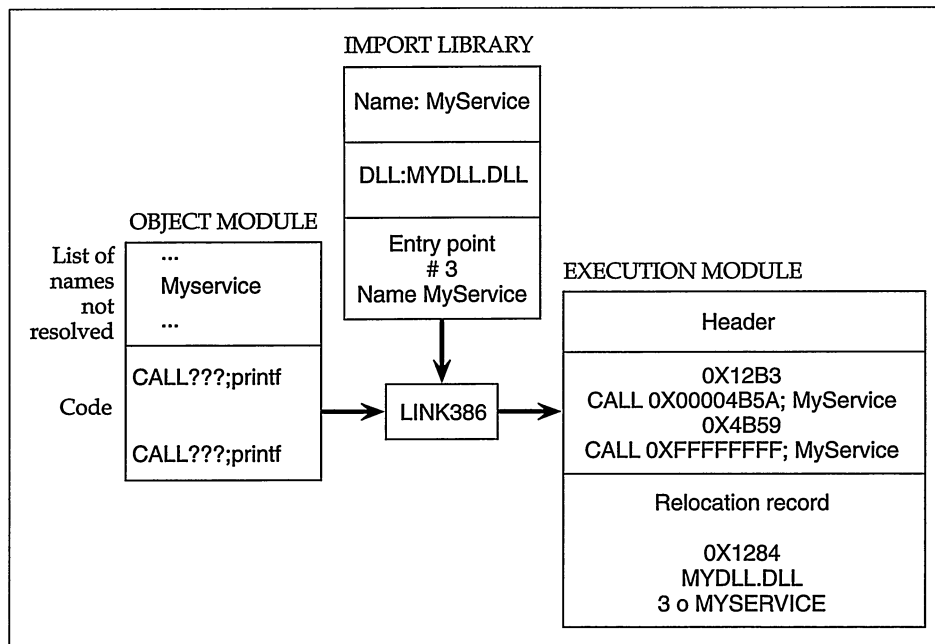
- The offset from the beginning of the code block wherein the call to a DLL function appears.

- The name of the DLL containing the service.
- The entry point of the function in the DLL, indicated with its ordinal value or with its name (never both). In practice, the relocation record built by the linker in the application is identical to that specified with the `.IMPORTS` directive in the DEF file or the information present in an import library.

The creation of relocation records in an executable module by the linker is always in a one-to-one relationship; for each API call there will appear only one relocation record even if the service is called several times in the program. For example, just think of all calls to `WinSendMsg()`!

The linker will resolve each single call in the program to the same DLL service by means of a *linked list*. Corresponding to each call to the same function present in the code of a DLL, the linker adds the offset of the address of the next call, and terminates the sequence with the dummy value of `0xFFFF`. Figure 10.12 illustrates how the linker solves all external references to DLL functions.

Consequently, the EXE image produced by the linker does not correspond to the real situation at the next application run. For each function residing in a DLL there is a specific relocation record that will indicate the source in a linked list of references. The values contained in the various locations are dummy values, and do not have anything to do with the actual position in memory where the DLL service is to be



**Figure 10.12** Scheme for resolving external references to functions residing in a DLL.

found. The discovery of all these pieces of information are delegated to the system loader at the moment when the application is loaded and executed.

## *Executing a Program That Accesses a DLL*

It is not possible to execute a DLL directly. The only action that is allowed is its loading into memory. This operation is performed automatically by any application containing one or more specific relocation records indicating the usage of a corresponding number of DLL services. The system loader will examine the structure of the executable module and search for possible relocation records. Each one of these records will indicate the name of the DLL containing the function called in the code. If the indicated DLL is not already present in memory, the loader will load it. Instead, if it is present in memory, it will increment the DLL's usage count. Once this is over, it is easy for the loader to fix in the EXE all unresolved addresses representing the calls to the services in the DLL. Although this is the general scenario, there are a number of operations being performed at each stage, and they can have different characteristics according to the specific situations implemented in the system. Lets examine how a DLL can be loaded.

## *Loading a DLL Implicitly*

When the system loader encounters a relocation record, it must first find the file corresponding to the DLL. Generally, DLLs are parked only in the directory mentioned in the environment variable LIBPATH, specified in CONFIG.SYS. This set of directories represents the search space for the system loader.

The same DLL can be present in more than one directory indicated in LIBPATH; in this case it is the first one encountered that gets loaded. The loading operation considers all code and data segments that make up the DLL. If the DLL were already present in memory due to some former application, then the code segments that are already present in memory are reused, since they are read-only resources. The criteria followed for data segments is different. It is possible that each instance of the DLL loads a new set of data, or each instance can always reference the same data segment. In the first case, data is specific to each single instance (*instance data segments*), while in the second case data is global to all instances (*global data segments*). The first solution is obviously simpler to implement, as well as the most convenient, although it is more demanding in terms of memory consumption. Soon we will examine which techniques to employ in order to implement the two different solutions.

Once the DLL is loaded into memory, the physical address of the various functions called in the application and residing in the DLL become known. The system loader will take care of changing the internal dummy references generated by the linker when creating the EXE, and replace them with the real values present in memory at that time.

The first operation is to get the name of the service called by the application. This information is expressed numerically or by means of the full name of the service. In the case of a number, it is compared with the DLLs *entry table*: That's where the segment of the DLL containing the service is defined, together with an offset to the

starting point of the block. Then the loader only needs to provide the application with an appropriate pointer for accessing the segment of the DLL that has just been loaded into memory. On the basis of this information and of the offset value, it can then write the correct address of the service. If the service in the application's relocation record is represented by its full name, then the loader must perform another operation—retrieve the ordinal number of the service in the DLL. To do this, it must examine the contents of a table of names present in the DLL's header. This second solution is therefore slower than the first one, and also requires more memory.

## Loading a DLL Explicitly

In addition to the action performed by the system loader when executing an executable equipped with relocation records, there is another alternative where the application takes care of loading the DLL directly. The operations to perform are very simple, and are all supported by a well-designed API. The actual loading of a DLL takes place through the function *DosLoadModule()*:

```
#define INCL_DOSMODULEMGR
APIRET APIENTRY DosLoadModule( PSZ pszName,
                                ULONG cbName,
                                PSZ pszModname,
                                PHMODULE phmod) ;
```

<i>Parameter</i>	<i>Description</i>
pszName	Buffer for accommodating any possible error messages
cbName	Size of the buffer for error messages
pszModname	Name of the file containing the DLL
phmod	Address of the identifier containing the handle of the DLL (module)
<i>Return Value</i>	<i>Description</i>
APIRET	Return code of the function

In the third parameter you have to indicate the name of the file containing the DLL, including the full pathname (if necessary). In the fourth parameter the *DosLoadModule()* function returns the handle to the DLL, a fundamental piece of information for accessing the DLL later. The loading of a DLL, both explicit as well as implicit, implies the increment of the module's usage count, which is initially set to zero. When the DLL is released, the corresponding memory is freed only when the usage count returns to zero, by means of successive calls to *DosFreeModule()*:

```
#define INCL_DOSMODULEMGR
APIRET APIENTRY DosFreeModule( HMODULE hmod) ;
```

<i>Parameter</i>	<i>Description</i>
hmod	Handle of the DLL (module)
<i>Return Value</i>	<i>Description</i>
APIRET	Return value of the function

Each process that accesses a DLL must call *DosFreeModule()* in order to correctly manage the memory used by the module. Once the DLL has been loaded, the application will want to know the address of the services contained therein. Naturally, the call cannot be by name, but only by address. The address can be found by calling *DosQueryProcAddr()*:

```
#define INCL_DOSMODULEMGR
APIRET APIENTRY DosQueryProcAddr( HMODULE hmod,
                                   ULONG ordinal,
                                   PSZ pszName,
                                   PFN* pPFN) ;
```

<i>Parameter</i>	<i>Description</i>
hmod	Handle of the DLL (module)
ordinal	Sequential number of the function within the DLL
pszName	Name of the function being sought
ppfn	Address of the function in the DLL
<i>Return Value</i>	<i>Description</i>
APIRET	Return code of the function

When you know the name of the loaded DLL, it is possible to find its handle with *DosQueryModuleHandle()*:

```
#define INCL_DOSMODULEMGR
APIRET APIENTRY DosQueryModuleHandle( PSZ pszModname,
                                       PHMODULE phmod) ;
```

<i>Parameter</i>	<i>Description</i>
pszModname	Actual name of the DLL
phmod	Address of the identifier containing the handle of the DLL (module)
<i>Return Value</i>	<i>Description</i>
APIRET	Return value of the function

The opposite operation, that of getting to the name of a DLL starting with a handle, can be done through *DosQueryModuleName()*:

```
#define INCL_DOSMODULEMGR
APIRET APIENTRY DosQueryModuleName( HMODULE hmod,
                                     ULONG cbName,
                                     PCHAR pch) ;
```

<i>Parameter</i>	<i>Description</i>
hmod	Handle of the DLL (module)
cbName	Size of the buffer
pch	Array of chars containing the name of the DLL
<i>Return Value</i>	<i>Description</i>
APIRET	Return code of the function



The following code fragment illustrates the techniques to follow when loading a DLL explicitly into an application:

```

...
static HMODULE hmod ;
...
switch( msg)
{
    case WM_CREATE:
        DosLoadModule( NULL, 0L, "MIE01.DLL", &hmod) ;
        if( !hmod)
        {
            WinAlarm( HWND_DESKTOP, WA_ERROR) ;
            ...
        }
        ...
        break ;

    case WM_CLOSE:
        DosFreeModule( hmod) ;
        break ;

    case WM_COMMAND:
    {
        PFN pfn ;
        ...
        DosQueryProcAddr( hmod, 0L, "MieClientColor", pfn) ;
        ...
    }
    ...
}
...

```

Naturally, it is necessary for the programmer to know not only the name of the required service, but also its syntax. The call to the function will be made through its pointer, followed by the parameter list:

```

...
( *pfn)( hwnd, ulLong) ;
...

```

The sample application will call a function that requires a window handle and a parameter of type ULONG.

## *Coding a DLL*

The absence of a traditional entry point is worth thinking about. Often a DLL is simply a collection of unrelated services. However, almost invariably, there will be a need for some preliminary operations before loading the DLL, in addition to other *clean-up* actions after its use. To meet these needs, the code in a DLL can be enriched with the

function `_DLL_InitTerm()`, which is called automatically by the system when the library is loaded and released. The syntax of `_DLL_InitTerm()` requires only two parameters:

```
ULONG _System _DLL_InitTerm( HMODULE hmodule, ULONG ulFlag );
```

<i>Parameter</i>	<i>Description</i>
<code>hmodule</code>	Handle of the DLL
<code>ulFlag</code>	Flag indicating the activation (0) or deactivation (1) of the DLL
<i>Return Value</i>	<i>Description</i>
ULONG	A nonzero value to indicate that either the initialization or the termination routine has ended successfully, zero to indicate a failure

The body of a DLL is essentially based on the examination of the `ulFlag` parameter in a switch statement:

```
...
ULONG _System _DLL_InitTerm( ULONG hmodule, ULONG flag)
{
    switch( flag)
    {
        case 0: // initialize
            ...
            break ;
        case 1: // terminate
            ...
            break ;
    }
    return 1L ;
}
...
```

The call frequency of the `_DLL_InitTerm()` function is controlled by directives that can be given in the module definition file, corresponding to `LIBRARY`. The possible choices are pairs `INITINSTANCE-INITGLOBAL` and `TERMINSTANCE-TERMGLOBAL`. The choice of a `xxxINSTANCE` directive implies that the function `_DLL_InitTerm()` is executed at the loading/releasing of the DLL for each process that accesses the DLL. On the other hand, the directive `xxxGLOBAL` implies the execution of `_DLL_InitTerm()` only when the DLL is physically loaded into memory for the first time, or when the last process using it will release it. The default combination is given by `INITGLOBAL` and `TERMGLOBAL`.

---

## Creating a Sample DLL

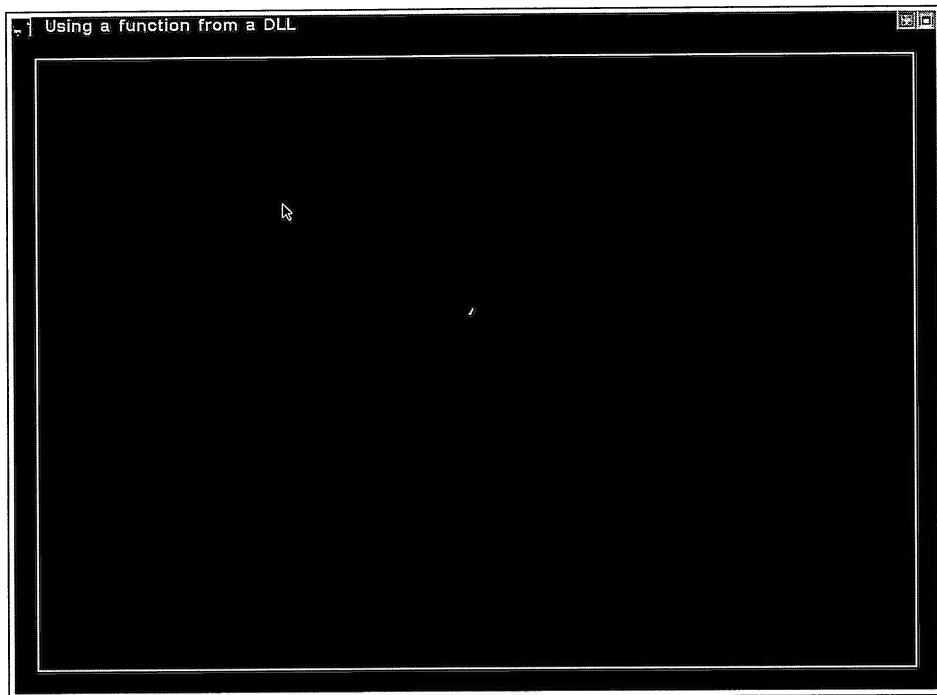
The best way to understand the nature of a DLL is to create one from scratch. In Figure 10.8 you can see the general scheme; the similarities to the development of an OS/2 executable are evident.

In this first sample, you will create a DLL named MIE01.DLL, specializing in providing some support services for applications that will be described in the next chapters. MIE01.DLL contains a service called *MieColorClient()* which will color the client window of an application and then draw a thin white border around the rectangle representing the client. The internal logic of *MieColorClient()* is very similar to many other samples of this book: Each time the window is resized or acquires focus, it will color itself with a different color. In Figure 10.13 you can see the output produced by the program, and its source code appears in Listing 10.4. The source code of MIE01.DLL is in Listing 10.5.

The service *MieColorClient()* is called in the code portion handling the WM\_PAINT message. The syntax of this service is:

```
COLOR EXPENTRY MieColorClient( HWND hwnd,
                               HPS hps,
                               PRECTL prc ) ;
```

The first parameter is the handle of the *client window*, followed by the handle to the presentation space returned by *WinBeginPaint()*. The syntax then requires a pointer to a structure of type RECTL. The return value of the function is the color currently assigned to the application's client window.



**Figure 10.13** Coloring the application's client window is delegated to a service residing in the MIE01.DLL dynamic linking library, developed specifically for this purpose.

With this information, *MieColorClient()* can color the client, and then display the white border, five pixels wide, entering from the maximum size of the client. Drawing the border is delegated to the function *WinDrawBorder()*, which is part of OS/2's API. In MIE01.DLL there is also the function *MieEmphasizeBorder()* which will emphasize a window when the *window context menu* is displayed, or when a *drag & drop* operation is performed.

## Compiling a DLL

Once the source code MIE01.C is written, you can compile it. As anticipated, this is an ordinary compilation that simply relies on setting a flag in order to inform the compiler the presence of C runtime library functions, and of the DLL nature of the source. During the linking phase, the DLL's module definition file is also put to good use.

## The DEF File of a DLL

The module definition file of a DLL performs a crucial role in the operations rendered by the linker. First of all, the `LIBRARY` directive replaces the usual `NAME`. With `LIBRARY` you tell the linker not to look for a *main()* function as the application's entry point. Furthermore, the code segments of the DLL will always be of the `PRELOAD` type, while the scenario gets somewhat more convoluted with respect to an ordinary PM application as far as the `DATA` directive is concerned.

By definition, a DLL is a set of compiled and linked functions that can be called simultaneously by more than one application. The presence of specific DLL APIs in the system responds to this organization. This means that the same service can be called from different processes, all running simultaneously, and it implies that you have to write reentrant code. This means you must be careful about managing data (identifiers with static storage class cannot be used), and about dynamic memory allocation performed by the code. There are two viable solutions for handling the data segment. Each single instance of a DLL can load the data segment provided that the modules were created according to the scheme that involves multiple copies for each instance (*instance data*); or one same data segment can be shared among all instances (*global data*). Figure 10.14 illustrates the two possibilities. Without considering the overall memory requirements, both solutions have pros and cons.

In the first case, you must specify the option `NONSHARED` corresponding to the `DATA` directive in the DEF file; in the second case, the option `SHARED`. The different instances of a DLL will then, respectively, not share and share the data segment, and all values therein contained.

The first solution is simpler and easiest to implement when you're first approaching DLL development, since it allows each single process using DLL services to maintain a private copy of all data. For example, you could run two instances of the application listed in Listing 10.8, and change the current color: You can see the result in Figure 10.15.

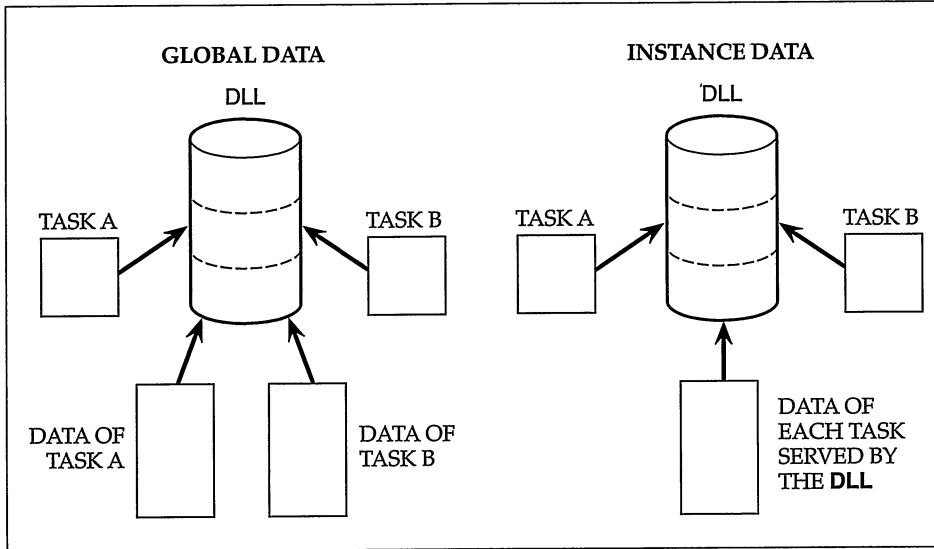


Figure 10.14 Instance data and global data for DLLs.

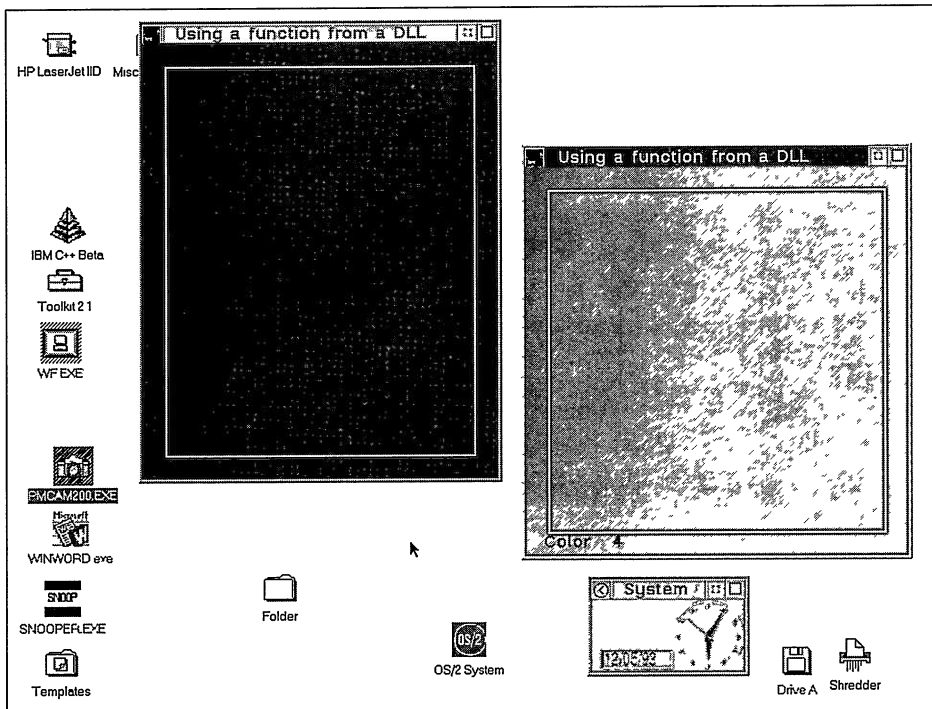


Figure 10.15 Execution of two instances that use the same services in the MIE01.DLL DLL.

If, instead, MIE01.DEF had the SHARED corresponding to the DATA directive, then the data segment would be the same for any instance of the DLL. To illustrate this new situation, run two copies of the application; each instance increments its private usage count. The situation is represented in Figure 10.16 which shows the numeric difference between the two colors used in the two application instances.

The activation of the window on the left side of the screen will make it color, and, differing from what is shown in Figure 10.14, it will take on the next color with respect to the previous active instance (Figure 10.17).

In addition to SHARED and NONSHARED there are also the options SINGLE and MULTIPLE, which all affect the linker's behavior when creating a DLL.

The option MULTIPLE forces all data segments of a DLL (both the *automatic data segment*, as well as any other), to be specific for each instance of the DLL. With SINGLE, on the other hand, the *automatic data segment* is shared among all instances of the DLL; the same is true for all other data segments of the DLL. In general, the default values for a DLL are SINGLE and SHARED.

The *automatic data segment* (ADS) is the combination of a set of logical segments, specialized in containing certain types of data. In `_DATA`, for example, you will find the *source file scope* identifiers, or those with static storage class, while `_BSS` will contain the noninitialized `static` identifiers. For an application written in the C language, the ADS will also contain the stack in the `_STACK` area. For a DLL there is no `_STACK` portion, since it will use the stack of the calling application.

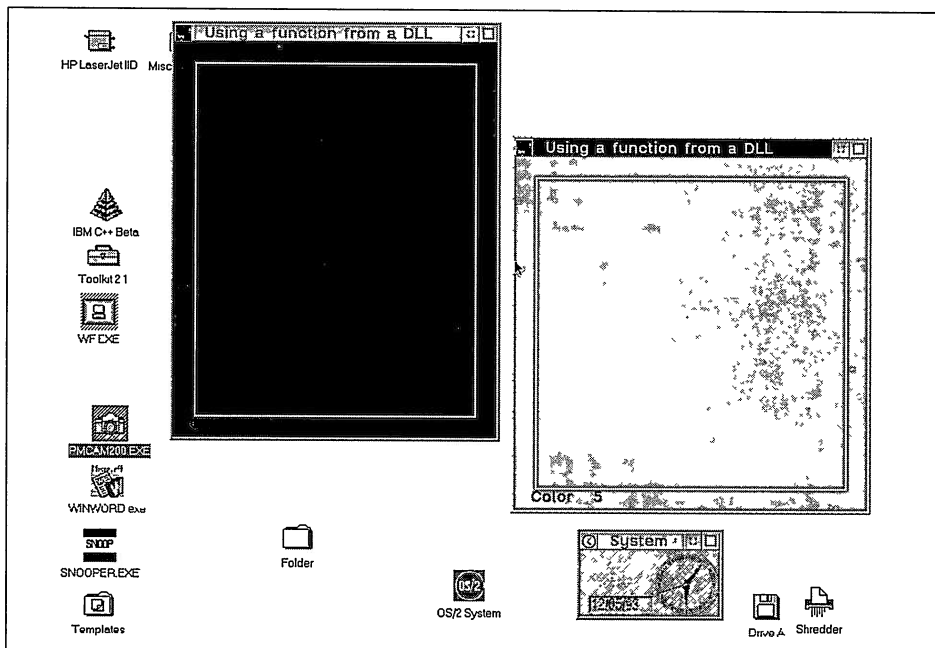
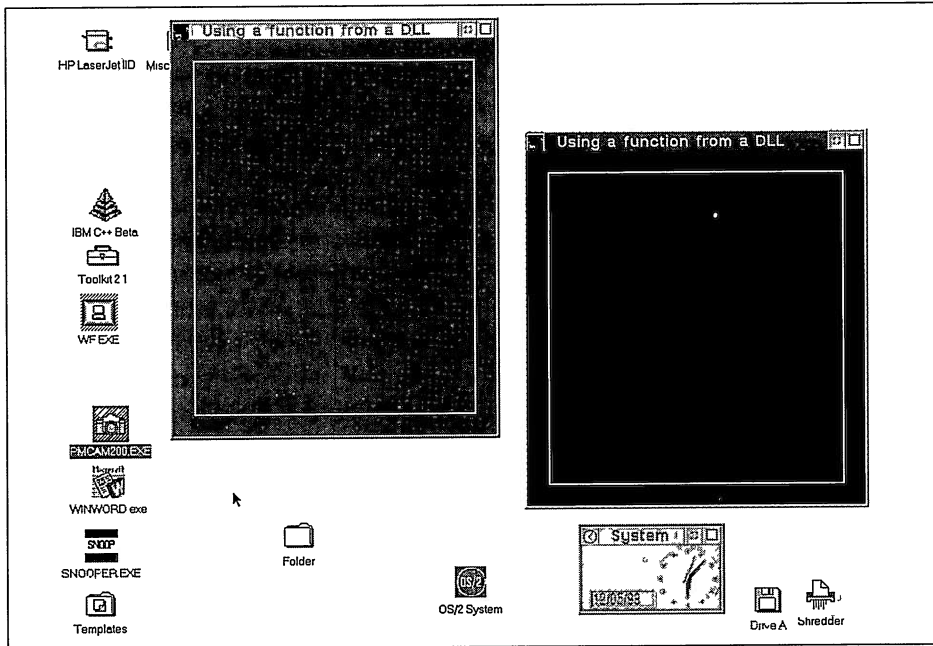


Figure 10.16 Execution of two instances that use the same services in MIE01.DLL, but with a shared data segment.



**Figure 10.17** When focus is transferred back to the first instance, the color selected for repainting the client is the one following the one used for the previous instance.

When implementing a simple DLL featuring a single data segment, using `SINGLE` or `SHARED` is irrelevant, just like `MULTIPLE` and `NONSHARED`. Instead, when a DLL has several data segments, you can have different combinations of the instance attributes (`SINGLE` and `MULTIPLE`) and of the sharing attributes (`SHARED` and `NONSHARED`), so that the DLL can have some segments of the *global* type, and others of the *instance* type. Table 10.2 summarizes all possible cases.

To create an application with several segments, you only need to follow the guidelines given in Chapter 9. There the subject is mainly code segments rather than data segments, but the same approach is applicable to data, and you can resort to segmented code—splitting the source code into several files. As we have seen in Chapter 9, the `SEGMENTS` directive in the module definition file allows you to diversify the nature of a segment with respect to what is established at a more general level with `CODE` or `DATA`.

Therefore, it is possible to develop a DLL where the ADS is of the *instance* type, while all other segments are *global*, or vice versa, according to your specific needs. Consequently, the combination

```
DATA MULTIPLE SHARED
```

will force the DS to be specified for each single instance of the DLL (*instance*), and any other possible data segment will be *global*. Naturally, the behavior of the *automatic data*

**Table 10.2 Interaction Among the Options That Control the Management of Data Segments in a DLL, Concerning Both the Automatic Data Segment As Well As Other Segments**

<i>Instance</i>	<i>Sharing</i>	<i>ADS</i>	<i>Other Segments</i>
-	-	Global	Global
-	SHARED	Global	Global
-	NONSHARED	Instance	Instance
SINGLE	-	Global	Global
SINGLE	SHARED	Global	Global
SINGLE	NONSHARED	Global	Instance
MULTIPLE	-	Instance	Instance
MULTIPLE	SHARED	Instance	Global
MULTIPLE	NONSHARED	Instance	Instance

*segment* cannot vary, nor can its status be changed, with the SEGMENTS directive, which is applicable only for the remaining group of data segments.

As anticipated, in the DEF file there will be no STACKSIZE directive, simply because a DLL does not have any private stack, but relies upon the stack of the calling application.

The services offered by a DLL become “visible” (externally) by listing them after the EXPORTS directive in the DLL’s module definition file. In the EXPORTS section, there will appear both functions of the EXPENTRY kind (like *window procedures* or *dialog procedures*) and procedures that resort to the `_optlink` calling convention.

Naturally, even a DLL can call functions residing in other DLLs. Services (if any) of the EXPENTRY kind which are called by the program but reside in another application appear after the IMPORTS directive.

The sample DLL in Listing 10.4 uses the OS/2 API service `WinDrawBorder()`, though it does not have an IMPORTS section in the module definition file. This absence is justified by the presence of the OS2386.LIB *import library* on the linker’s command line.

## Producing an Import Library

Rather than writing a long list of all services residing in various DLLs and callable from the source code—in this example, the MIE01.DLL DLL—it is much more practical and convenient to generate an *import library*. The operation is simple and takes advantage of the IMPLIB.EXE utility. The command line syntax for invoking this program requires you to specify two parameters, corresponding to two files. You must give a name to the *import library*—generally the name of the DLL followed by the extension of LIB. The second parameter required by IMPLIB is the name of the DEF file from which you wish to extract the names of the services listed in the EXPORTS



section. The action of `IMPLIB` is simply to produce an object file with the extension `LIB` by reading what is listed in the `EXPORTS` section of a `DEF` file:

```
IMPLIB MIE01.LIB MIE01.DLL.DEF
```

Then the resulting *import library*—like `OS2386.LIB`—must always be indicated when linking any application using the services of `MIE01.DLL`. The command line invoking the linker in `MIE01.MAK` looks like this:

```
link386 mie01,,, os2386 mie01, mie01
```

The tool for managing library modules within `WorkFrame/2` will work correctly even on an import library. Therefore, it is possible to create a kind of *super import library* which collects all information regarding services living in various DLLs; this is exactly what has been done with `OS2386.LIB`. A last word of caution about writing DLLs: Once all compilation and linking operations have been performed, remember to transfer the DLL file into one of the library directories on the `LIBPATH`. You can leave the DLL in its original directory only if that directory is listed by `LIBPATH` (set `LIBPATH=.;C:\OS2\DLL;...`).

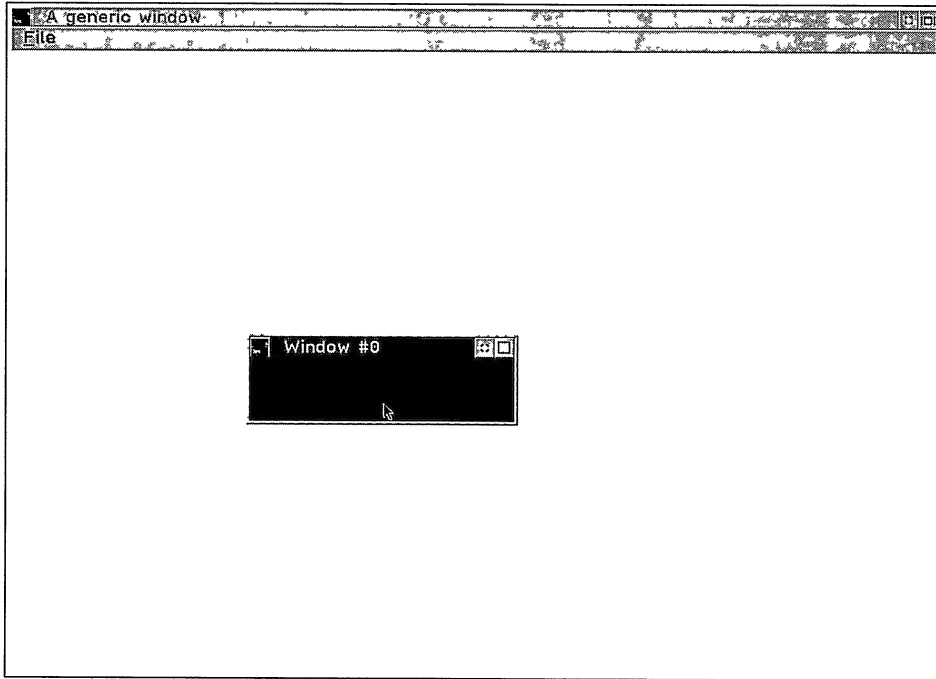
---

## Creating a New Control

Let's now see how you can generate a new class of windows. Nothing prevents you from registering the class directly in a DLL—rather, it is a convenient and sound approach that even the associated window procedure be inside the same module. In Listing 10.6 you can see the source code of the `WINDOW` application that will create a window of the class `TWENY`. This class is registered in `MIE01.DLL`, enriched with yet more services in addition to the function `InitTweny()` for registering the class, and `TwenyWndProc()`, its window procedure. `TwenyWndProc()` will catch only a limited number of messages: Almost all of the message flow will go directly to `WinDefWindowProc()`. The only peculiarity of this class is that of changing its color when the user clicks the left mouse button. It is not much of a feat, but it is good enough to demonstrate how the whole works. The application's output is shown in Figure 10.18, while the source code of both `WINDOW` as well as of `MIE01.DLL` is listed in Listing 10.5, while Listing 10.6 refers to an example application, `WINDOW`, which exploits those services.

The class `TWENY` does not stand out for any special functionality, but it is a good starting point for creating other customized applications. `TWENY` is visible only to `WINDOW`, and to any other application that directly calls the function `InitTweny()` to register the class. The concepts of subclassing and superclassing described in the first part of this chapter, together with the registration of a class inside a DLL, turn out to be extremely useful for producing applications. The software designer can collect in special DLLs new classes of specialized windows, capable of performing special operations for individual programs. Imagine, for example, that you need to generate a screen form for data input. The specification wants you to request the social security number, which is always a fixed character sequence. You could force the *entryfield* to





**Figure 10.18** The child window belonging to a window class registered in the MIE01.DLL DLL.

be that number of characters long, simply by using PM APIs. However, at least for now, PM does not offer any provision for limiting the type of characters that can be accepted by an *entryfield*. You could therefore use a DLL to register a new class of window, based on the features of a `WC_ENTRYFIELD`, and then intercept in the window procedure (among many others) the messages `EM_DEFINPUT` and `WM_CHAR`. The first message is defined by the user as being specific to this class and defining which keys can be accepted as input. The information passed along will be useful when intercepting the message `WM_CHAR` to verify that the input given by the user is consistent with the desired limitations.

The greatest advantage of this solution is that you can perform input validation directly when it is being generated, and thereby you avoid canceling input while cleaning up the *entryfield*'s contents on the screen and requesting that the user provide correct data anew. This approach also greatly simplifies applications for inexperienced users.

---

## Creating a New Predefined Class

Let's now consider a poorly documented—at least for the time being—aspect of PM programming, but one which is critical for building advanced PM applications. This

way of writing code also allows a high degree of flexibility and modularity of applications. We are speaking about the construction of new window classes within appropriate DLLs.

PM's API features a set of predefined classes: OS/2's is typically equipped with fifteen of them. Therefore, you can create a listbox window in your own application without any need for registering the class or writing any special window procedures, since both operations have been carried out beforehand by the designers of OS/2 who placed the various WC\_ classes in the DLLs that are loaded when the system is booted. These DLLs are always present in the system while OS/2 is working, and therefore can be accessed by any application.

The solution shown in Listing 10.7 starts by registering and defining a window class for a new class in a DLL; but it forces the application that needs it to have at minimum a relocation record, so that it can load the DLL. When creating a customized control, the approach is somewhat different: The DLL must always be present in memory, and the application building windows belonging to the new class does not have to call any function present in the DLL. Figure 10.19 illustrates the scheme to follow when registering new customized classes.

By following this approach it is possible to define new kinds of windows, encapsulating all functionality within a DLL, and then distribute them. Other software designers will be able to access a series of services having the same form as those provided by the API of PM, albeit originating from a third-party source. In Listing 10.10 you can see the source code of a generic application called USECTL that creates a child window, the client window of which belongs to the class TWENY.

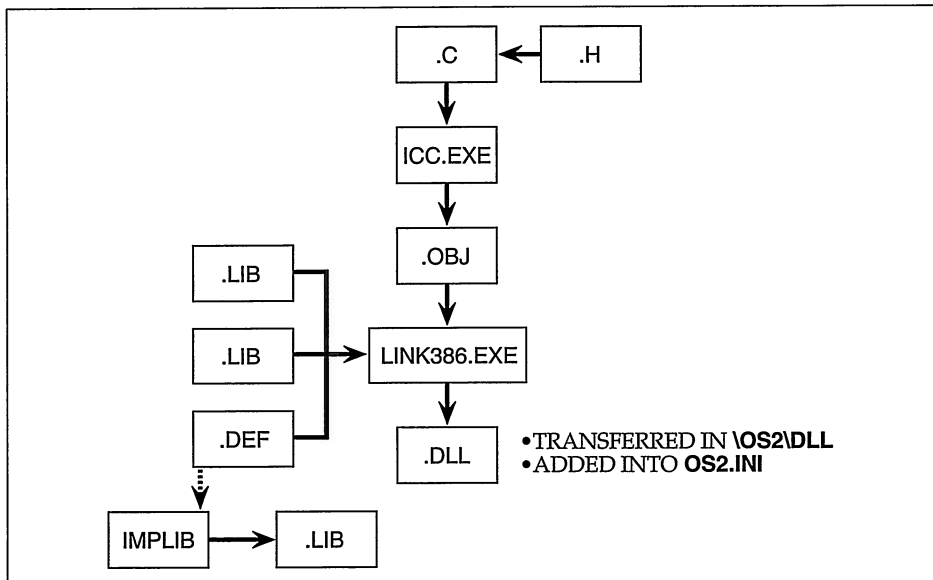
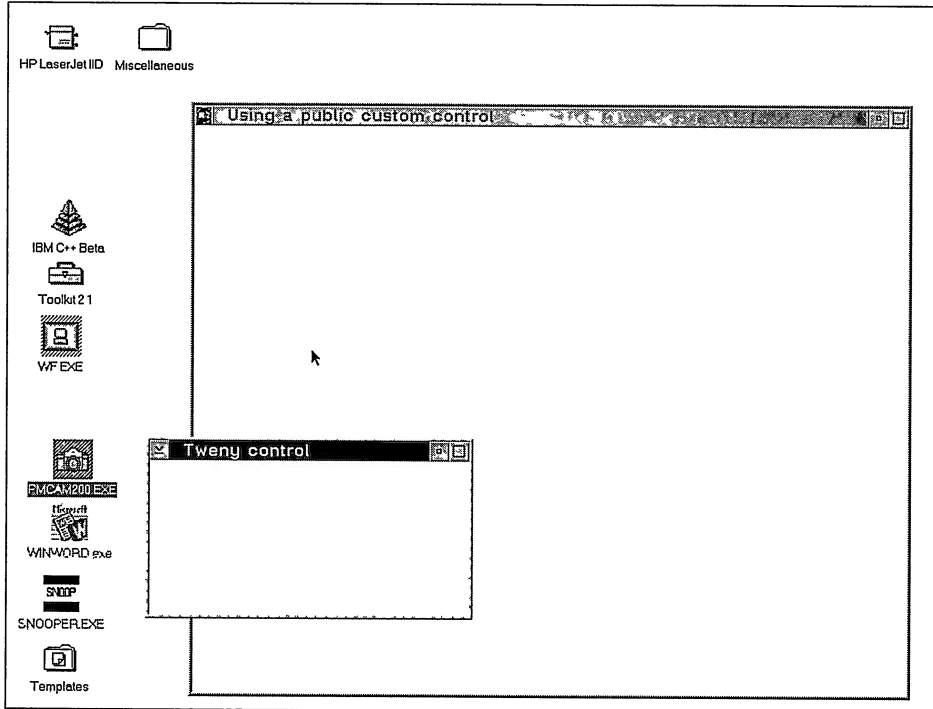


Figure 10.19 Scheme to follow when implementing new window classes extending the standard outfit of OS/2 PM.



**Figure 10.20** The client window of the child window belongs to the class TWENY, which is registered in the MIECC.DLL DLL and loaded by OS/2 during bootup.



The registration of this new class takes place in the MIECC.DLL DLL (Listing 10.8 and 10.9), which is loaded by OS/2 during its initial boot, and which is always present in memory (Figure 10.20). Each application calling *WinCreateMsgQueue()* within its own code will also be able to access the services of MIECC.DLL.

## Constructing a New Window Class

In order to define a window class, as we have seen in the preceding chapters, it is necessary to register the window class and then create an appropriate window procedure. In the case of a customized control, all this takes place in a DLL, where the registration is placed in a separate function with respect to the window procedure. There are no special limitations concerning the name of the function that registers the class. Considering that the DLLs in OS/2 2.1 always have an initialization/termination function called *\_DLL\_InitTerm()* it is often convenient to place the registration code in this function.

In this way, you make sure that PM will call the *TwenyWndProc()* function when the DLL is loaded: This ensures that the class TWENY will be registered from the very

beginning and remain as long as the system is alive. Here you can see the code fragment that actually registers the class TWENY:

```

...
ULONG _System _DLL_InitTerm( ULONG hmodule, ULONG flag)
{
    switch( flag)
    {
        case 0: // initialize
            if( !WinRegisterClass( NULLHANDLE, "TWENY",
                                   TwenyWndProc,
                                   CS_SIZEREDRAW | CS_PUBLIC,
                                   sizeof( ULONG)))
                WinAlarm( HWND_DESKTOP, WA_ERROR) ;
            break ;
        case 1: // terminate
            break ;
    }
    return 1L ;
}
...

```

In the class registration call, the flag `CS_PUBLIC` must be set: this is the only place where it can be used without generating a runtime error (in Listing 10.5, although registration took place in a DLL, it was not possible to specify the `CS_PUBLIC` flag).

The first parameter of `WinRegisterClass()` is set to `NULLHANDLE` since there is no anchor block handle available. The class's window procedure, on the other hand, is always a function of the type `EXENTRY` because it is called, as usual, directly from PM, and it can have any name you please. The window procedure must be exported in the `EXPORTS` section of the DLL's `DEF` file:

```

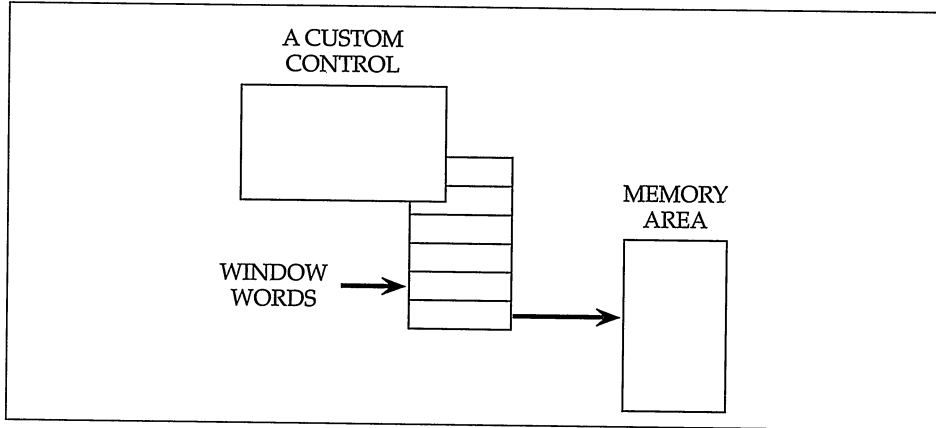
...
EXPORTS
    TwenyWndProc
...

```

Access to this function does not ever happen as a direct call by naming it explicitly. This means that no application will have a relocation record referring to the DLL containing the custom control. Access to this code is nonetheless granted by the special way the DLL is loaded, as we will see shortly.

## Writing the Window Procedure

This is not the first time you have dealt with a window procedure. In this case, though, it takes on a special meaning, since this is a code fragment invisible to applications, just as is the case of the `WC_` classes. `TwenyWndProc()` intercepts all messages pertaining to the new class, like a generic window procedure. However, as this is a window procedure akin to those of the predefined classes, it must also deal with the problem



**Figure 10.21** Information management in a window class acting as a custom control.

of interacting with the owner and parent windows of each control. Therefore, it is necessary to devise a set of notification codes, messages, styles, and possibly other data structures to allow programmers to use the class with the same ease with which they can take advantage of `WC_MLE`, `WC_CONTAINER`, or other predefined classes. When writing the window procedure, you must keep in mind the possible interactions with the owner/parent window, and implement some mechanism by passing the `WM_CONTROL` message with specific notification codes.

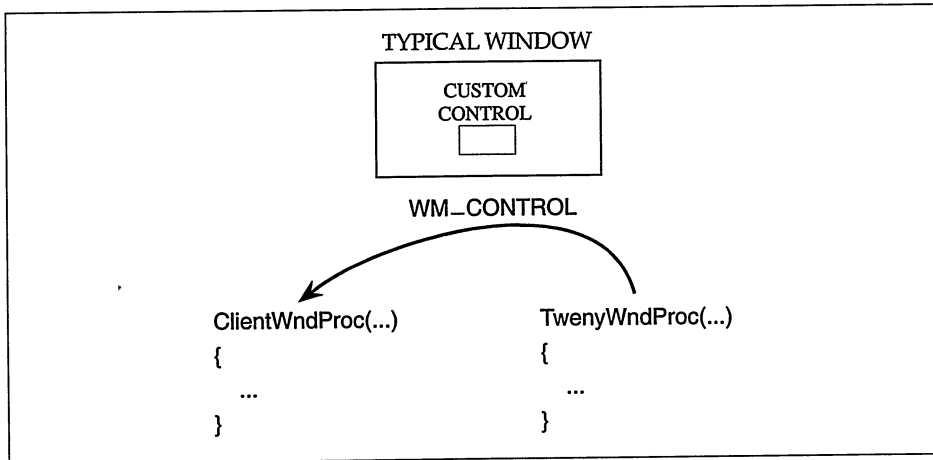
In the window procedure it is not wise to adopt static storage class identifiers. In this case each window of this class would reference data directly associated with the code, instead of encapsulating them with the window handle. You would need to use many *window words*, as you can see in Table 7.3, on predefined classes. An alternative and/or complementary solution is to keep in a few bytes the size of the window words, and then store the information associated with each window class in memory blocks or sub-blocks dynamically allocated. Figure 10.21 illustrates the whole setup.

In this example, a simple interaction with the window class `TWENY` is pressing the left mouse button; you will change the window's color. The information contained in the window words is constantly updated after each mouse click. The owner of `TWENY` class window will be notified about the event through the message `WM_CONTROL`, which contains the notification code `TN_CHANGE`, as shown in Figure 10.22.

The applications that use the services of `TWENY` find out what the current active color is at any time by sending the message `TM_QUERYCOLOR`. This message and the previous notification code are both defined in `MIECC.H`.

## *Creating a Window of the New Class*

The construction of a window of the class `TWENY` follows the same standard rules we have seen in the previous chapters for creating any ordinary window. You can apply



**Figure 10.22** Scheme for sending the notification code `TN_CHANGE` to the owner window of a control of the class `TWENY`.

a `WS_style`, even if in this particular case of `TWENY` no special style has been set for this new class (considering the limited number of operations supported).

If the control's look is to be customized through specific style flags, you must make sure that the new defines given in the header file associated with the class are actually placed in the lower word of the long value defining the `WS_style`s. The call to `WinCreateWindow()` in the following code fragment will create a control of the class `TWENY`:

```

...
hwndTweny = WinCreateStdWindow(  HWND_DESKTOP, 0L,
                                &ulFCFrame,
                                szControlName,
                                "Tweny control",
                                0L,
                                NULLHANDLE,
                                0L,
                                &hwndClient) ;
...

```

It is fundamental to remember to assign an ID to the window, in order to be certain than any possible notification codes get passed correctly to the owner window. The developer has to call `WinSetWindowUShort()` with `QWS_ID` as the second parameter.

## Passing Information

The last parameter of the function `WinCreateWindow()` is a pointer to a data area allocated by the application filled in with information that is specific to the class to which the new window belongs. This information can be accessed in the class's

window procedure by responding to the message `WM_CREATE`, automatically generated by the function that creates a window. Class `TWENY` is simple and limited, so there is no need to transfer special information when an instance of this class is created. For more complex controls, the situation is radically different: Figure 10.23 illustrates the mechanism you need to implement for passing information specific to a predefined class.

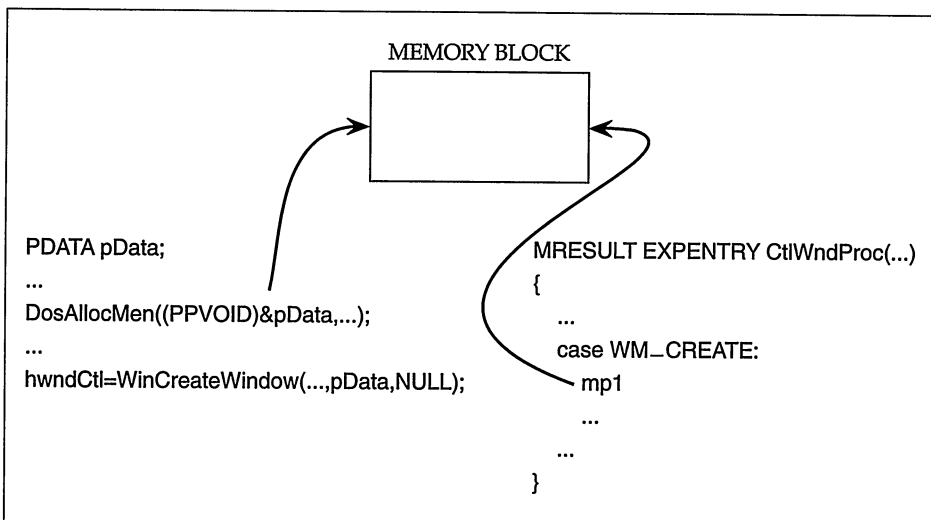
For each access to the function `TwenyWndProc()`, both the owner window's handle and the ID of the control are retrieved by interacting directly with the reserved memory area of the `TWENY` class windows. This operation is done outside of the `switch` statement, in order to make sure that you access the owner of each specific control:

```
...
// owner handle
hwndOwner = WinQueryWindow( hwnd, QW_OWNER );

// window ID
sID = WinQueryWindowUShort( hwnd, QWS_ID );
...
```

Alternatively, the request of the owner can be placed in the code fragment dealing with the messages that need it. Both solutions are valid, and it is not easy to decide which is best.

When the owner window (or even any other window in the application where the `TWENY` class control appears) sends the message `TM_QUERYCOLOR`, there is not really much work to do. The window words contain the currently selected color value used to paint the window background. This information is passed back to the calling function, bypassing the traditional `WinDefWindowProc()`:



**Figure 10.23** Usage of the parameter `pCtldata` in `WinCreateWindow()` for passing information specific to a predefined window class.





<i>Parameter</i>	<i>Description</i>
hini	Handle to an INI file
pszApp	Name of the application
pszKey	Key name
pszDefault	Buffer containing any possible error condition
pBuffer	Buffer containing the value retrieved from the INI file
cchBufferMax	Size of the pBuffer
<i>Return Value</i>	<i>Description</i>
ULONG	Dimension of the string returned in pBuffer

The purpose of this function is to retrieve a text string from a profile file, like OS2.INI. The first parameter allows you to indicate which file should be read: It can be any file handle or any of the defines listed in Table 10.3.

The second parameter, *pszApp*, identifies an array of CHAR containing the name of the application. Most often it is preferable to indicate NULL, as this will obtain a list of all application names present in the profile. In the case of the SYSDLLS application, it is possible to indicate directly the name of the entry that you want to search for, since it is always SYS\_DLLS.

With *pszKeyName*, the third parameter, you can indicate the key name of the item to search. The same rule just described still holds; by indicating NULL you will obtain a list of all key names contained in the profile. For loading a control declared in a DLL you must indicate the string *Load*.

The third parameter of *PrfQueryProfileString()* is an array of CHAR which is used by this function for containing any possible error message if the reading operation is unsuccessful. The last two parameters are a buffer for receiving the string and a LONG that defines its length. The return value of the function is the number of characters read. In Listing 10.8, the whole becomes like this:

**Table 10.3 The Constants for Reading Profiles with the *PrfQueryProfileString()* Function**

<i>Flag</i>	<i>Value</i>	<i>Definition</i>
HINI_PROFILE	(HINI) NULL	Performs the search in the user's (the application's) profile file, and, if there is no matching entry, will continue the search in the system's profile.
HINI_USERPROFILE	(HINI) -1L	Performs the search only in the user's profile file.
HINI_SYSTEMPROFILE	(HINI) -2L	Performs the search only in the system's profile.

```

...
PrfQueryProfileSize( HINI_PROFILE, "SYS_DLLS", KEY_NAME, &length) ;
DosAllocMem((PPVOID)&pBuffer, uLen,
             PAG_READ | PAG_WRITE | PAG_COMMIT) ;
PrfQueryProfileString( HINI_PROFILE, "SYS_DLLS", KEY_NAME,
                      "", pBuffer, uLen) ;
...

```

In `pBuffer` you get the string `PMCTLS` referring to the OS/2 DLL which contains all the predefined window classes. If this string has more than one entry they are separated by a blank character (" "). The whole string is divided into its subcomponents and each one becomes an item in the listbox. As the user presses the Add pushbutton what is written in the application *entryfield* becomes a new entry in the DLL listbox. Besides this external change, the pushbutton pressure triggers the update of the LoadPerProcess area in OS2.INI. `PrfWriteProfileString()` executes the writing in this user file:

```

#define INCL_WINSHELLDATA
BOOL WINAPI PrfWriteProfileString( HINI hini,
                                   PSZ pszApp,
                                   PSZ pszKey,
                                   PSZ pszData) ;

```

<i>Parameter</i>	<i>Description</i>
<code>hini</code>	Handle to a file INI
<code>pszApp</code>	Name of the application
<code>pszKey</code>	Name of the key
<code>pszData</code>	Buffer containing the value to write in INI
<i>Return Value</i>	<i>Description</i>
<code>ULONG</code>	Dimension of the string captured in <code>pBuffer</code> .

After modifying OS2.INI you must shutdown and restart the system. To add a DLL with a custom control you must follow these two rules:

- Transfer the DLL in \OS2\DLL
- Modify OS2.INI, section SYS\_DLLS—LoadPerProcess

Considering that you are not allowed either to erase or to copy a custom control DLL when properly installed (the file is in use by the system), the development phase is quite long and cumbersome. To shorten the development process it is advisable to implement a DLL without adding it to SYS\_DLLS—LoadPerProcess, instead loading it explicitly with `DosLoadModule()` (temporarily removing the `CS_PUBLIC` flag from the registration call). Once the DLL logic has proven its efficiency, you can easily convert it back restoring the `CS_PUBLIC` style.

## Some Considerations

Since the definition of a new window class always accessible to any application implies the modification of the contents of OS2.INI, it is important to perform the operation carefully; any mistakes in this file could actually prevent successive bootups from the hard disk. To avoid this unpleasant situation, keep in mind the following rules:

- Before making any change to OS2.INI, create a backup copy of this file. The boot drive always contains in the \OS2 directory a file named INI.RC, a kind of backup copy of OS2.INI which is handled automatically by the system. Once this file is securely saved, you can use the MAKEINI utility to generate a new copy of OS2.INI (there also exists a file named INISYS.RC corresponding to OS2INI.SYS).

```
[C:\]MAKEINI INI.RC
```

- Always keep handy the original installation disk of OS/2; without this floppy you're in trouble! Any change or replacement of OS2.INI can be done only if booting has been done from some source other than the hard disk containing the file. The installation floppy comes to your rescue here.

Here's the correct, complete order of operations:

- After booting from the floppy, go into \OS2 and make a copy of OS2.INI on another floppy.
- Reboot the system directly from the disk and add the entry LoadPerProcess in the SYS\_DLLS section of OS2.INI, specifying the name MIECC or any other DLL of your own.
- Reboot the system and check that MIECC has actually been loaded correctly (a simple PM program calling *WinQueryClassInfo()* is good enough for testing for the presence of the new class).

In the case of problems, booting from the floppy and then restoring the old OS2.INI are the only viable solutions. All these rules are even more important if you have installed the HPFS file system; with the old FAT you can bypass the problem with a simple bootup from a floppy or with the dual boot, if supported.

# *Data Sharing Techniques: Clipboard and DDE*

One of the most interesting aspects of the user interface of OS/2 is its data sharing capabilities among several applications. Passing of information among programs written by different software houses makes the user's life a lot easier, and reduces the importance of the so-called integrated software packages. Each single application in PM specializes in performing a set of very specific functions (Lotus Ami Pro is a superb word processor, while 1-2-3 can handle and display numeric data very easily). The transfer of data processed by one program simply means that you can take better advantage of the system as a whole and achieve a significant increase in personal productivity.

In OS/2, there are two tools that allow for information sharing among applications: the Clipboard and the DDE (Dynamic Data Exchange) protocol. The two mechanisms have some points in common, but differ substantially at the design level. The Clipboard is a data-sharing tool that is completely user driven, while DDE depends essentially on the design logic of an application (application-driven).

---

## **The Clipboard**

In OS/2, there is a block of memory handled by the system's memory manager that is able to contain the information to be shared among different applications. This part of the memory manager is known as the Clipboard. Hence, the Clipboard is not a program, but a way of handling the problem of data sharing. Nothing prevents the name Clipboard from being assigned to some application whose only purpose is that of displaying the contents of that memory area. As a matter of fact, in the Productivity folder you will find the Clipboard viewer application.

The sharing of information through the Clipboard is totally under the user's control. The first operation to be performed is selecting the data within an application and then transferring it to the Clipboard. This operation is the equivalent of "publishing" the set of information by putting it in the special memory area. It is always the user's responsibility, the within same application or in any other application, to make a local copy of the data.

**Table 11.1 Clipboard Operations**

<i>Operation</i>	<i>Description</i>
Cut	Transfers the selected object from the application to the Clipboard.
Copy	Performs a copy of the selected object toward the Clipboard.
Paste	Retrieves, from the Clipboard, a copy of the object and pastes it into the current application.
Delete	Destroys the selected object without affecting the contents of the Clipboard.

According to this example, when all manual operations are done, the system will contain three copies of the same object: the original one, the one contained in the Clipboard, and the one generated by the local copy operation.

The terminology used by PM to describe the typical phases involved in using the Clipboard is universal. The process is known as *cut & paste*, while the single actions are described in Table 11.1.

According to the CUA specifications, the actions described in Table 11.1 are associated with specific accelerator keys, as listed in Table 11.2 and illustrated in Figure 11.1.

In Figure 11.2 you can see the logic governing the operations of copying and pasting an object to/from the Clipboard.

## *Clipboard Management*

There are well-defined rules that govern the use of the Clipboard in OS/2 systems. First, the user may transfer into the Clipboard only one object at a time. Thus, this memory area contains only one graph, image, or text block at a time. The insertion of new data into the Clipboard will automatically destroy what was previously there.

The object transferred into the Clipboard is classified according to various formats, called *clipboard data formats*. OS/2 supports three standard data formats for representing information in the Clipboard: text, bitmap, and metafile. In addition to these, which are adequate for most situations, it is possible to define at the API level yet other formats better suited for the special needs of applications.

**Table 11.2 Accelerator Key Combinations Associated with the Cut & Paste Operations Supported by the Clipboard**

<i>Accelerator</i>	<i>Action</i>
Ctrl+Ins	Copy
Shift+Ins	Paste
Shift+Del	Cut
Del	Delete

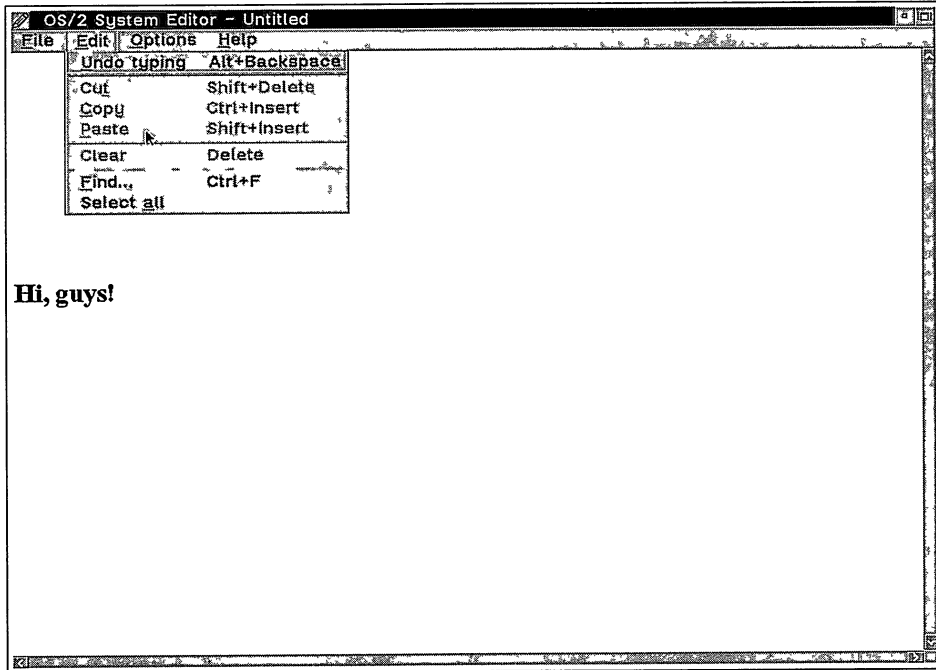


Figure 11.1 The look of a typical Edit menu in a PM application supporting the Clipboard conventions.

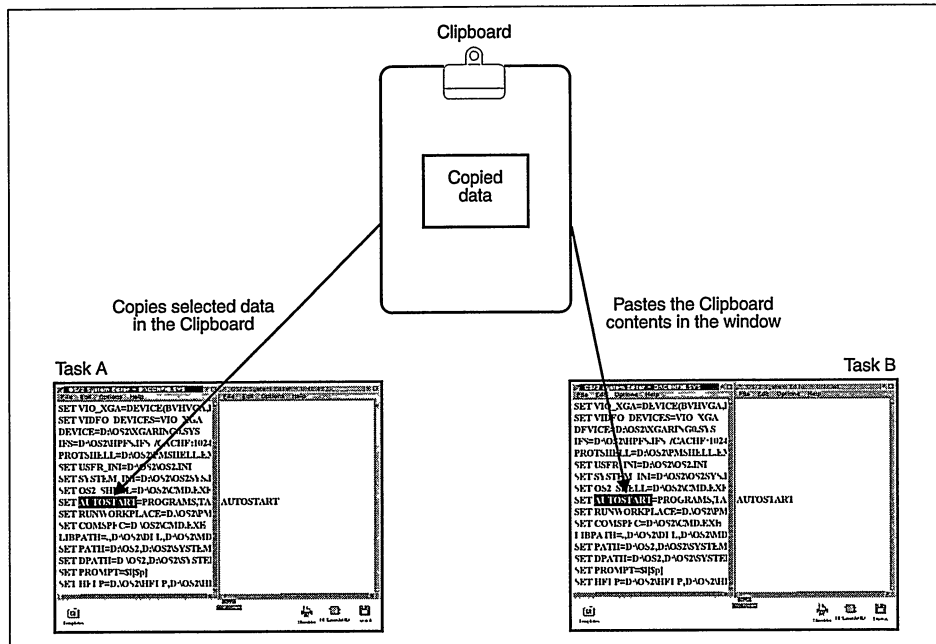


Figure 11.2 Object transfer to and from the Clipboard.

## Inserting Data into the Clipboard

The first operation to perform when transferring information into the Clipboard is opening it. This is done with the function *WinOpenClipbrd()* which does not return any value until the current thread is not allowed to access the Clipboard. This does not mean, however, that the application trying to access the Clipboard will remain locked up. The possible presence of pending messages in its own message queue is detected by the *WinOpenClipbrd()* function that will allow them to be handled. The syntax of *WinOpenClipbrd()* is the following:

```
#define INCL_WINCLIPBOARD
BOOL APIENTRY WinOpenClipbrd( HAB hab ) ;
```

<i>Parameter</i>	<i>Description</i>
hab	Handle of the anchor block
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

Neither the return value, nor the sole parameter to this function will give the application a handle for accessing the Clipboard. Only the success of the call will signal that you have access to the Clipboard. Once this phase is over, you will have to empty the Clipboard of its previous contents by calling *WinEmptyClipbrd()*. In that way, you make sure that any format describing any object previously contained in the Clipboard is reset.

```
#define INCL_WINCLIPBOARD
BOOL WinEmptyClipbrd( HAB hab ) ;
```

<i>Parameter</i>	<i>Description</i>
hab	Handle of the anchor block
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The transfer of an object to the Clipboard is obtained by calling *WinSetClipbrdData()* and indicating the data format employed.

```
#define INCL_WINCLIPBOARD
BOOL APIENTRY WinSetClipbrdData( HAB hab,
                                  ULONG ulData,
                                  ULONG fmt,
                                  ULONG rgfFmtInfo ) ;
```

<i>Parameter</i>	<i>Description</i>
hab	Handle of the anchor block
ulData	Handle or pointer to the object being transferred to the Clipboard
fmt	Clipboard format
rgfFmtInfo	Data type: any value in Table 11.4



**Table 11.3 The Standard Formats Used in Describing the Nature of the Object Transferred into the Clipboard**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
CF_TEXT	1	The object is in text format.
CF_BITMAP	2	The object is a bitmap.
CF_DSPTEXT	3	The object is text but with custom format.
CF_DSPBITMAP	4	The object is a bitmap but with custom format.
CF_METAFILE	5	The object is in metafile format.
CF_DSPMETAFILE	6	The object is in metafile custom format.
CF_PALETTE	9	The object is in palette format.

<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The first parameter of this function is the handle of the application's anchor block, as originally returned by the call to *WinInitialize()*. With the second parameter, *WinSetClipbrdData()* passes to the Clipboard a reference to the object being copied. Most often, this will be a handle or a pointer to some memory area containing the object. The formats supported by *WinSetClipbrdData()* are listed in Table 11.3.

The last parameter informs *WinSetClipbrdData()* as to the type of the second one: The available alternatives are: CFI\_HANDLE or CFI\_POINTER for memory models, and CFI\_OWNERDISPLAY and CFI\_OWNERFREE for the information contained in the Clipboard (Table 11.4).

It is also possible to indicate NULLHANDLE as the second parameter of *WinSetClipbrdData()* to implement a delayed rendering of the information that qualifies the

**Table 11.4 The Standard Formats Supported for Transferring an Object into the Clipboard**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
CFI_OWNERFREE	0x0001	The owner of the object transferred to the Clipboard must take care of freeing the handle.
CFI_OWNERDISPLAY	0x0002	The format of the object is defined by the owner of the Clipboard.
CFI_POINTER	0x0400	The second parameter of <i>WinSetClipbrdData()</i> is a pointer to a block of memory containing the information to be transferred into the Clipboard.
CFI_HANDLE	0x0200	The second parameter of <i>WinSetClipbrdData()</i> is a handle to a memory area containing the information to be transferred into the Clipboard.

object. This solution is particularly useful when the description of the data transferred into the Clipboard is application proprietary. Using `NULLHANDLE` in place of a handle or a pointer also implies that the application becomes the owner of the Clipboard through the function `WinSetClipbrdOwner()`:

```
#define INCL_WINCLIPBOARD
BOOL APIENTRY WinSetClipbrdOwner( HAB hab, HWND hwnd ) ;
```

<i>Parameter</i>	<i>Description</i>
hab	Handle of the anchor block
hwnd	Handle of the window owing the Clipboard
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The system will allow the operation only one window at a time, simply because it relates the window with the object that is theoretically present in the Clipboard (but since you indicated `NULLHANDLE`, the Clipboard contains no data). The need to become owner of the Clipboard is flagged by the receipt of the message `WM_RENDERFMT` any time an application attempts retrieving the object that is theoretically present in the Clipboard. Only at this time will the transfer to the Clipboard of the handle or of the pointer to the object actually take place, by calling `WinSetClipbrdData()`. Once the writing of data into the Clipboard has been done, it is necessary to release it by means of `WinCloseClipbrd()` in order to allow other applications to access it.

```
#define INCL_WINCLIPBOARD
BOOL APIENTRY WinCloseClipbrd( HAB hab ) ;
```

<i>Parameter</i>	<i>Description</i>
hab	Handle of the anchor block
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

## *Transferring an Object*

The main function of the Clipboard is making an object public: This implies that the information contained therein should be accessible to any application. Therefore, it is necessary that the information reference a shared memory block or the applications accessing the Clipboard will not be able to build a valid reference for getting to the data. To this end, it is necessary to distinguish between passing a handle and passing a pointer. In the first case, the Clipboard itself will be responsible for sharing the passed bitmap or metafile handle. The application providing the handle loses completely any reference to the object, because the Clipboard will become its sole owner. In the case of a pointer, the allocation of the memory block must be achieved through `DosAllocSharedMem()`, setting the flag `OBJ_GIVEABLE`.

## Retrieving the Clipboard's Contents

An application interacting with the Clipboard will first open it in order to make sure that no other program will interfere in the operation. When this condition is met, the application must determine which formats are currently employed by the Clipboard in representing the object.

The application calls the function *WinQueryClipbrdFmtInfo()* to determine if the format of the object in the Clipboard corresponds to the supported standard. The adoption of this function is appropriate here, since it does not imply the former opening of the Clipboard, as you can infer from its syntax:

```
#define INCL_WINCLIPBOARD
BOOL WINAPI WinQueryClipbrdFmtInfo( HAB hab,
                                     ULONG fmt,
                                     PULONG prgfFmtInfo) ;
```

<i>Parameter</i>	<i>Description</i>
hab	Handle of the anchor block
fmt	Data format
prgfFmtInfo	Address of a memory area containing the formats used to represent the data present in the Clipboard

<i>Return Value</i>	<i>Description</i>
ULONG	Handle to the data retrieved from the Clipboard

In addition to the handle to the anchor block—which can be replaced by a simple `NULLHANDLE`—the syntax requires that the user indicate what format is expected, and the address of a `ULONG` wherein the function can return the information qualifying the type of description of the object present in the Clipboard (`CFI_POINTER` or `CFI_HANDLE`, for instance).

With *WinQueryClipbrdData()* the program can ask the Clipboard if a particular format is available at any given moment.

```
#define INCL_WINCLIPBOARD
ULONG WINAPI WinQueryClipbrdData( HAB hab, ULONG fmt) ;
```

<i>Parameter</i>	<i>Description</i>
hab	Handle to the anchor block
fmt	Data format

<i>Return Value</i>	<i>Description</i>
ULONG	Handle of data retrieved by the Clipboard

The return value of this function is a handle to the data expressed in the required format (or a pointer), or `NULLHANDLE` if the request cannot be honored. For text—the simplest data sharing format—the value returned by *WinQueryClipbrdData()* is a pointer to the memory area containing data. It is necessary that whatever is contained in the Clipboard also be copied to some local memory area in order to allow the application to access it even after that the Clipboard is closed. In fact, once the

Clipboard is closed, the application is not allowed to access any object contained therein while it is the full owner of the local copy.

## Examining the Contents of the Clipboard

All OS/2 systems are equipped with the Clipboard viewer application, the purpose of which is to display in a window the contents of the Clipboard. Writing a similar application is a simple task that only requires the implementation of the procedures for reading the contents of the Clipboard.

PM's API offers the pair of functions *WinQueryClipbrdViewer()* and *WinSetClipbrdViewer()*, respectively, for getting and setting the window that will examine the contents of the Clipboard. Any task can access the Clipboard by means of *WinOpenClipbrd()* and then examine its contents. The action performed by *WinSetClipbrdViewer()* is different because the code receives the message WM\_DRAWCLIPBOARD whenever the contents of the Clipboard are changed. This solution also allows other applications to interact with the Clipboard without interfering with the standard mechanism for inserting data.

Only one window in the whole system can operate as a *Clipboard viewer*, but it does not have to be the owner of the Clipboard. In the code handling the message WM\_DRAWCLIPBOARD, the Clipboard viewer will interface with the Clipboard in a way similar to any other application supporting *cut & paste* operations. The examples in Listings 11.1 and 11.2 refer to the development of two distinct applications that, respectively, transfer information to the Clipboard, and read from the Clipboard. Transforming CLIPSHOW, the program that reads the contents of the Clipboard, into a Clipboard viewer is a simple operation.

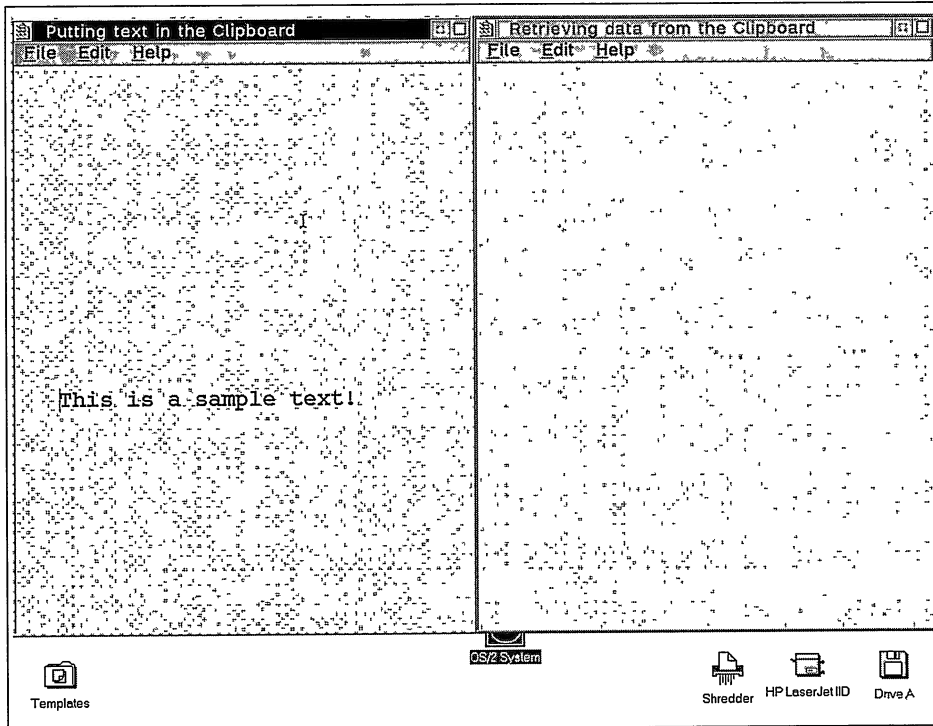


## CLIPPUT and CLIPSHOW

To clarify the rules for interacting with the Clipboard, let's examine two simple applications, CLIPPUT and CLIPSHOW. CLIPPUT will transfer into the Clipboard a text string typed into a WC\_MLE class window, while CLIPSHOW will display that string in its client, after appropriate menu item interactions (Figure 11.3). Actually, the WC\_MLE class has some specific messages to handle cut & paste operations. I did not use them to illustrate a more generic way to interact with the Clipboard.

In CLIPPUT, the Copy option of the Edit menu is enabled only if the *mle* acting as the application's client window truly contains some text. To verify this condition, the application intercepts the notification code MLN\_CHANGE generated by all windows belonging to the WC\_MLE class, and conveyed by a WM\_CONTROL message. It is also a good practice to check if the contents of the text window are non-blank. The user might have pressed some keys and then undone that input with the backspace key. This solution, however, is not efficient. The search for the MLN\_CHANGE notification code inevitably slows down the whole program without offering a correct view over the input provided by the user.

It is much better to intercept the message WM\_INITMENU generated by the system whenever the focus is transferred to a top level menu of the menu bar. This allows for



**Figure 11.3** The look of CLIPPUT and CLIPSHOW is bare just for exemplifying the interaction rules between the application and the Clipboard.

easier handling of the dynamic change of state of the menu items Copy, Paste, Delete, and Cut. A left mouse button-click over the menu bar or the pressing of the ALT key triggers this message.

In the WM\_INITMENU case branch, the presence of text in the *mle* is checked for by *WinQueryWindowText()*. The buffer filled in by the function will contain a true copy of whatever the user typed into the input window. The dynamic change of the Copy menu item in the Edit menu is obtained simply by sending the message MM\_SETITEMATTR with the right combination of flags (initially, Copy is always disabled). Figure 11.4 shows that Edit:Copy is enabled when the *mle* contains some text.

In order to transfer to another application whatever object has been produced and copied to the Clipboard with CLIPPUT, no special program is needed. For example, the system editor, E.EXE, can prove that the criteria followed by CLIPPUT for interfacing with the Clipboard is correct (Figure 11.5). Let's examine the logic governing this operation by analyzing the code of CLIPSHOW.

CLIPSHOW does not perform any operation other than that of displaying the contents of the Clipboard, provided it is in the CF\_TEXT format. In the menu template of the resource file, the Paste menu item under Edit is disabled, due to the attribute

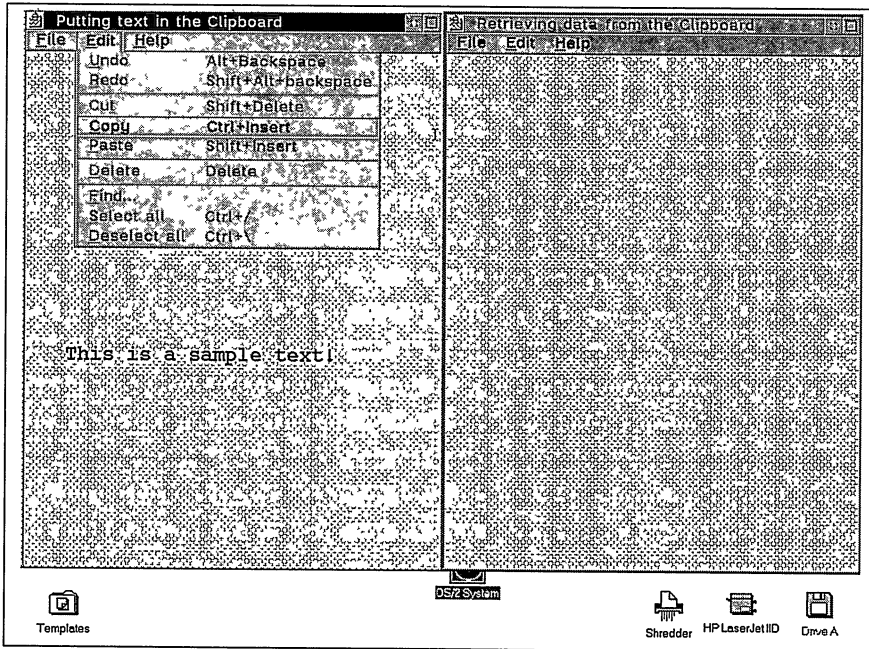


Figure 11.4 The Copy menu item is enabled when text is typed into the *mle*.

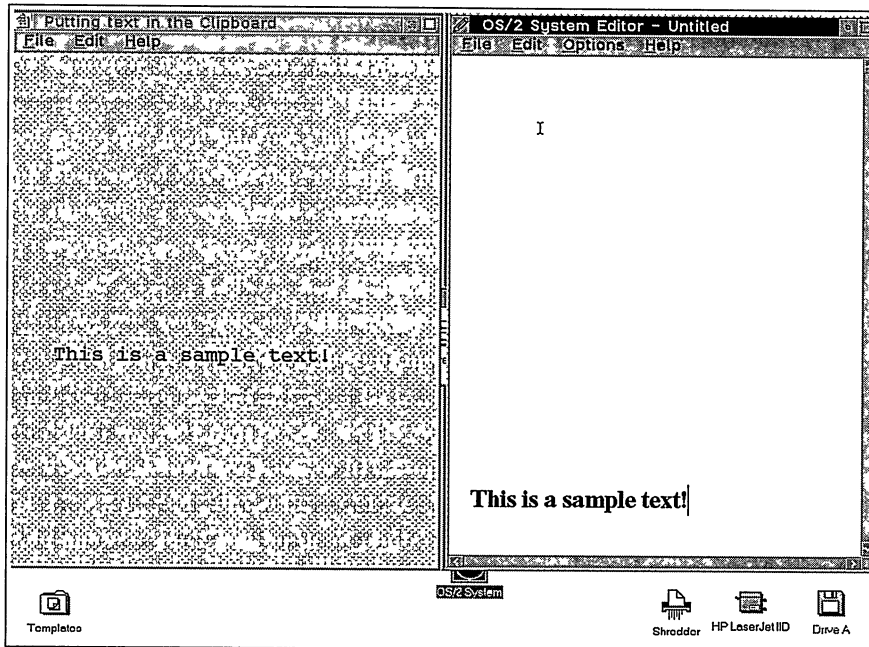


Figure 11.5 The system editor will display the text typed into the *mle* of CLIPPUT.

MIA\_DISABLED. Even in this case, the possible interaction with the Clipboard takes place when the message WM\_INITMENU is received.

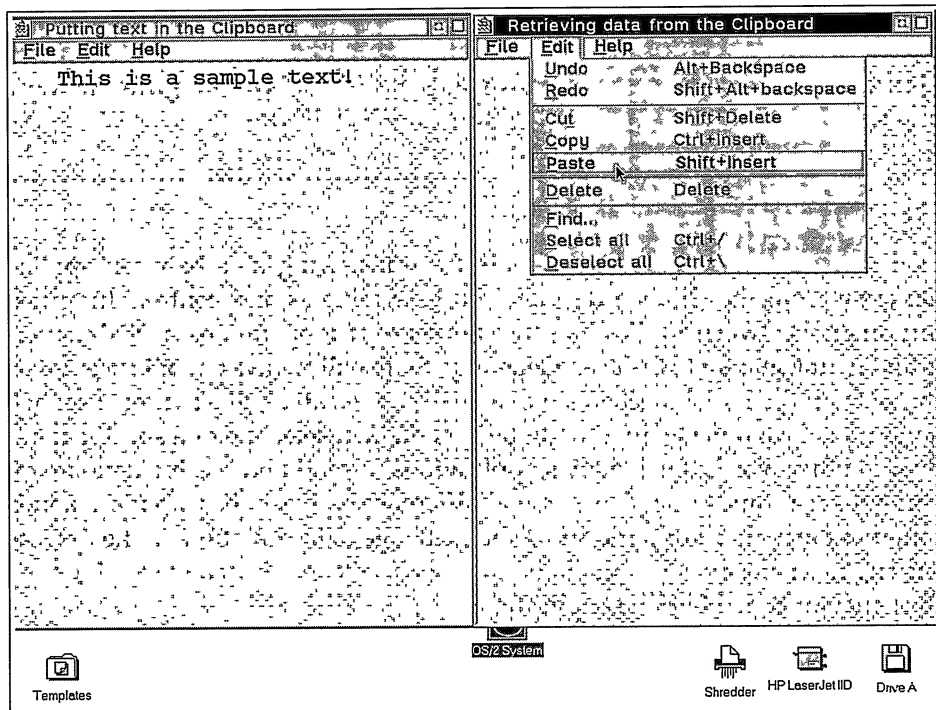
On the basis of the information obtained with *WinQueryClipbrdFmt()*, CLIPSHOW takes care of modifying the attributes of the Paste menu item under Edit (Figure 11.6). The retrieval of the Clipboard's contents is delayed until the message generated by the selection of Paste is actually received.

The text retrieved from the Clipboard appears in the *mle* of CLIPSHOW. The sequence of operations performed consists of:

- Opening the Clipboard
- Retrieving a pointer to the area containing the text

The text present in the Clipboard is then copied to a local buffer, an array of CHAR specifically declared in the program. Once this phase is over, you can close the Clipboard and release the shared memory area with *DosFreeMem()*. The operation is more truly that of decrementing a usage count: The system will then take care of the actual disposal.

The logic governing the output of the text is concentrated in the code handling the message WM\_PAINT, in order to refresh the window at any moment (Figure 11.7).



**Figure 11.6** In the Clipboard, there is an object described by the format CF\_TEXT: CLIPSHOW that can enable its own Paste menu item.

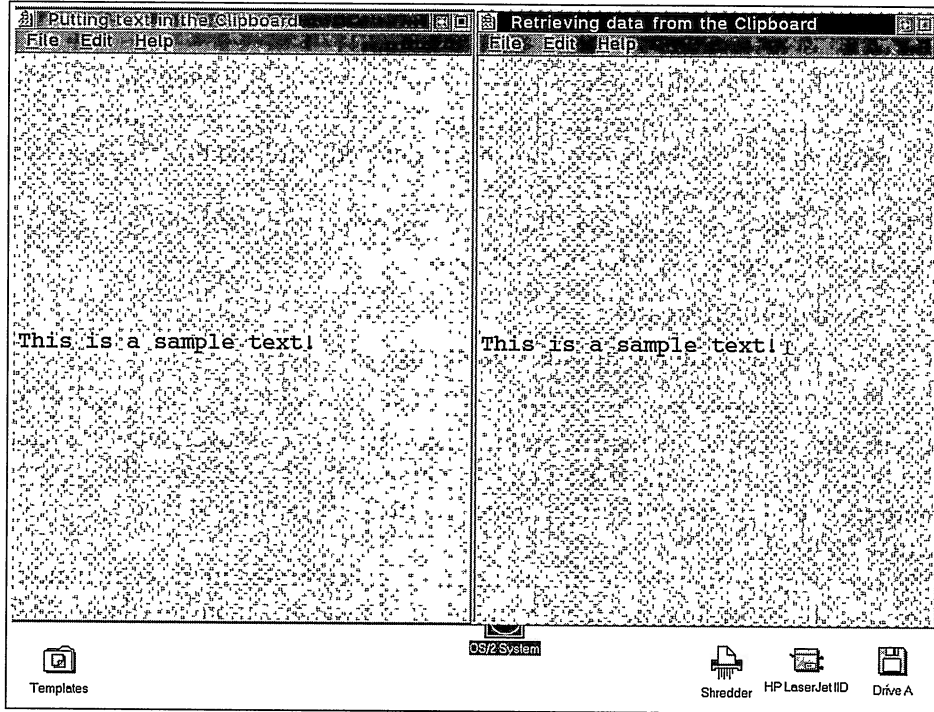


Figure 11.7 The actual text string displayed with E.EXE in Figure 11.5 now appears at the center of the client window of CLIPSHOW.

## Dynamic Data Exchange

DDE, just like the Clipboard, is a tool for transferring data between applications. Different from the Clipboard where the user initiates and controls the passing of information between applications through *cut & paste* operations, a DDE conversation is totally transparent to the user. Generally, there will always be some user interaction that sets up the initial mechanism of a DDE conversation, like a direct user action or an indirect program event: It might be a menu item selection or a specific time of the day. In other words, DDE allows applications to send and receive data on an asynchronous real-time basis. The essential difference between the Clipboard and DDE is found in the main actor performing the exchange of information. In the first case, the sharing mechanism is dependent on the user, while in the second case it depends on the behavior coded into the applications.

The DDE protocol has been developed by many companies, although Microsoft was the first to implement it in MS Windows, during the mid-1980s. DDE is a public-domain communication protocol implemented according to a set of rules that involves sending and receiving specific messages. Table 11.5 summarizes the ten messages that characterize DDE for PM.



**Table 11.5 The Messages That Characterize the DDE Communication Protocol**

<i>Message</i>	<i>Value</i>	<i>Description</i>
WM_DDE_INITIATE	0x00A0	Issued to initiate a DDE conversation.
WM_DDE_INITIATEACK	0x00A9	Issued to acknowledge initiation of a DDE conversation.
WM_DDE_REQUEST	0x00A1	Issued to request data.
WM_DDE_POKE	0x00A6	Issued to pass data.
WM_DDE_ADVISE	0x00A4	Issue to set up a permanent link on a specific item
WM_DDE_UNADVISE	0x00A5	Issued to terminate a permanent link on a specific item
WM_DDE_DATA	0x00A3	Issued to transfer data.
WM_DDE_ACK	0x00A2	Issued to acknowledge receipt and comprehension of a message by the receiving application.
WM_DDE_EXECUTE	0x00A7	Issued to execute some action in the receiving application.
WM_DDE_TERMINATE	0x00A8	Issued to terminate a DDE conversation.

In order to understand how to use the DDE messages, let's imagine a typical scenario: A PM communication program retrieves some stock quotations from an information service bureau.

By means of a DDE conversation initiated by Lotus 1-2-3 2.0, the communication program sends to a work sheet the closing price of the selected stocks. The work sheet employs the data in two ways: to update the cells of a work sheet containing the profile of an investment portfolio and to change a graph built on that data. Changes in the work sheet cells will trigger DDE transactions in a second DDE conversation. The work sheet transfers the contents of a set of cells to a PM word processor that will place the updated data in a client portfolio appearing in a table of a financial report. The final outcome presents to the user a real-time image of the stock portfolio through the graph, while the work sheet maintains the most recent quotations and the financial report document is always up to date.

Consequently, DDE operates like a bridge between different applications by providing a well-defined mechanism for transferring data, and constitutes yet another level of integration. DDE opens the way for a set of applications developed by third parties that are capable of being integrated with existing ones, and that enhance the overall productivity of the user.

Programs supporting DDE must operate in the PM screen group. PM applications are distinguished from OS/2 applications at the *kernel* level, mainly because of the presence of a message queue. The messages are generated by PM in response to events that happen in the system. PM applications differ in architecture from those at the

kernel level also because they do not use the three subsystems Vio, Kbd, and Mou in order to handle input or output. Instead, they receive messages from PM that indicate when the user presses a certain key or the mouse button. Therefore, they also send a message to PM whenever they need to display data on the screen. The same applies to PM applications that support DDE, since all DDE activities are performed by sending and posting PM messages to other applications. When a program transfers data to another by means of DDE, it will actually pass a reference to a shared memory block as one of the message components.

## A DDE Conversation

DDE is a tool for transferring data among applications. Since most of the DDE activities are under the control of the program (and not under the user's control, as with Clipboard), this is a protocol that governs how and when data are to be passed among applications. It is not a public memory handler like the Clipboard.

The main subject of DDE is the *conversation*: Each time a program resorts to DDE, it starts a conversation consisting of one or more DDE transactions. The application that initiates the conversation is known as the DDE *client*, while the application that provides the data is called the DDE *server*. In a single conversation, a client requests data of a server. An application can also be engaged in more than one conversation at a time. Since it will always be the client that initiates a conversation, a DDE server that requests data from another application must, in turn, initiate another conversation wherein it plays the role of a client. Therefore, an application can be both a client and a server at the same time, though in two different conversations.

## Designing a DDE Conversation

A DDE conversation always takes place between two PM windows, one for each program (it would not make sense to implement a DDE application within one application, although it is feasible). Each window is identified by its handle, which is also one of the components of the DDE message. The code handling a DDE message is placed in the window procedure of the program's main window, even though it is much more convenient to create an invisible window of a different class for each DDE conversation. This way the flow of messages is separated from the origin. The DDE conversation relies on a three level hierarchy for identifying a conversation.

- The application name (*application*)
- The topic name (*topic*)
- The item name (*item*)

The client specifies the name of an application when the conversation is initiated. Sometimes it omits it and invites any other program to engage in the communication. The topic name provides a logical context for each conversation. If omitted by the client, the conversation may cover *any* topic. Finally, each item must have a unique name. The three defines (application, topic, and item) have a specific meaning within

a spreadsheet. Lotus 1-2-3 is the application, the work sheet SHEET.WG2 is the topic, and the cell range A1:B3 is the item.

Often the development of a DDE server or client software is required to support any application and any item. On the other hand, when you implement vertical applications, the interaction between client and server is focused on a specific item.

The purpose of a DDE conversation is to provide the client with a series of information on the basis of specific requests. The conversation that takes place between the client and a server, after its activation, consists of *transactions*. The basic requirement is setting up a conversation, which must then be followed by one or more transactions. There are six different kinds of fundamental transactions: DDE: REQUEST, ADVISE, UNADVISE, POKE, EXECUTE, and TERMINATE, each one having a corresponding message.

## *Features of a Transaction*

A DDE client requests the server for data at any given moment. The message WM\_DDE\_REQUEST performs this operation. Furthermore, the client also posts a WM\_DDE\_ADVISE to establish a link with the server on a certain item (for instance, the contents of a cell in a spreadsheet).

The transaction produced by WM\_DDE\_ADVISE has two distinct variations, known in jargon as *hot-link* or *warm-link*. In both cases, the client and the server set up a *permanent link* pertaining to some *item* of conversation. The two variations are distinguished only as far as the physical passing of information between the server and the client is concerned. In the warm-link the server notifies the client that a change has occurred in the value of the item involved in the permanent link. In a hot-link the new data is actually transferred. In both cases, the interaction between the server and the client takes place by sending the message WM\_DDE\_DATA.

To interrupt this kind of relationship between the client and the server, the client must send a termination request (WM\_DDE\_UNADVISE), as illustrated in Figure 11.8.

Furthermore, the client posts to the server (WM\_DDE\_POKE) any piece of data, whenever it is appropriate to do so. With such a solution, you can avoid activating a new conversation between two applications with inverted roles.

With the WM\_DDE\_EXECUTE message, the client passes to the server a text string corresponding to a set of executable commands for the receiving application. There are application specific actions, like opening a document, displaying an image, and so on. The termination of a DDE conversation can be brought about both by the client as well as the server, by sending the message WM\_DDE\_TERMINATE. The transaction must be performed by both parts; all other transactions must be performed by the client only. If an application acting as the server needs to receive data from another application, and thus behave like a client, it needs to initiate a new DDE conversation. Figure 11.9 shows the direction of message flow in the DDE protocol.

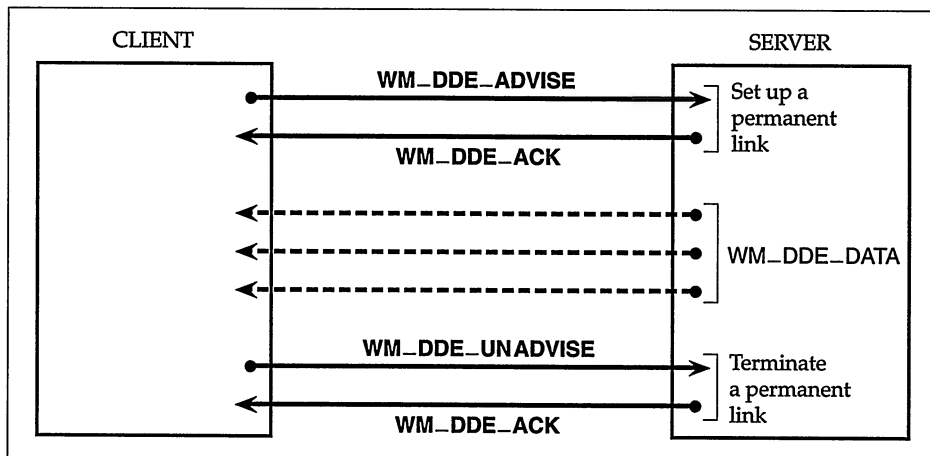


Figure 11.8 The establishment of a permanent link between a client and a server.

### Initiating a DDE Conversation

In order to initiate a conversation, the would-be client has to call the function *WinDdeInitiate()*. The “would-be” in the previous sentence is justified by the fact that there is no absolute certainty that the application calling *WinDdeInitiate()* will effectively find a server.

```
#define INCL_WINDDE
BOOL APIENTRY WinDdeInitiate(  HWND hwndClient,
                               PSZ pszAppName,
                               PSZ pszTopicName,
                               PCONVCONTEXT pcctxt) ;
```

<i>Client</i>	<i>Message</i>	<i>Server</i>
	WM_DDE_INITIATE	→
←	WM_DDE_INITIATEACK	
	WM_DDE_REQUEST	→
	WM_DDE_POKE	→
	WM_DDE_ADVISE	→
	WM_DDE_UNADVISE	→
←	WM_DDE_DATA	
←	WM_DDE_ACK	
	WM_DDE_EXECUTE	→
←	WM_DDE_TERMINATE	→

Figure 11.9 Direction of messages in a DDE conversation.

<i>Parameter</i>	<i>Description</i>
hwndClient	Handle of the window that will act as the client in the conversation
pszAppName	Name of the application sought for as a server
pszTopicName	Name of the topic desired for this conversation
pcctxt	Address of a CONVCONTEXT structure
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The handle refers to the issuing window, while the two text strings correspond to the name of the application being sought, and the conversation's topic, respectively. In the fourth parameter, the application has to indicate additional information, like the code page used (so that the conversation between client and server is not affected by problems deriving from system settings).

The DDE protocol not only has a set of specific messages, but also involves some highly specialized functions. *WinDdeInitiate()* plays a crucial role in DDE, since it will seek potential servers for a potential client. The action performed by this function is sending the message WM\_DDE\_INITIATE to all windows of WC\_FRAME class present in PM at that very moment (the *frame* then passes it to its client). This means that this function does not return any kind of value to the application until *all* windows have been explored in currently executing applications in PM. Figure 11.10 summarizes this situation.

Despite the rather different syntax with respect to that of *WinSendMessage()*, there are a number of similarities between these two functions. The entrance in the target window procedure requires the traditional data: the window's handle, the message WM\_DDE\_INITIATE, *mp1* containing the handle of the issuing window, and *mp2* containing the address of a DDEINIT structure:

```
typedef struct _DDEINIT
{ // ddei
    ULONG cb ;
    PSZ pszAppName ;
    PSZ pszTopic ;
} DDEINIT ;
```

The three members of DDEINIT should be self-explanatory. The first one defines the size, the second and the third are the corresponding parameters in *WinDdeInitiate()*. The DDEINIT structure is accommodated in a memory area specifically allocated by *WinDdeInitiate()* itself; actually, this is the first action performed. Then, *WinDdeInitiate()* issues the message WM\_DDE\_INITIATE to all frame windows having HWND\_DESKTOP as their parent. The following code fragment comes from a server window procedure:

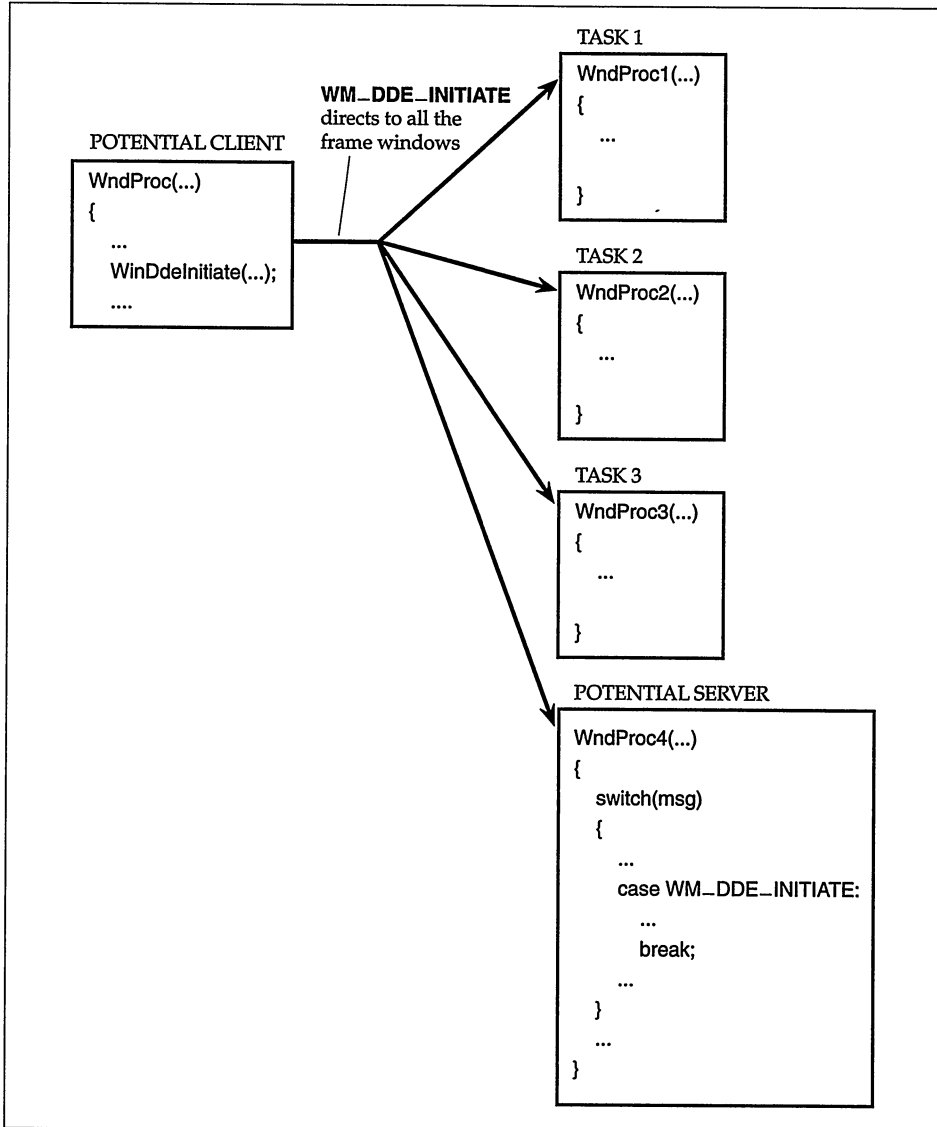


Figure 11.10 Scheme of action performed by *WinDdeInitiate()* during the initiation phase of a DDE conversation.

```

...
case WM_DDE_INITIATE:
{
    PDDEINIT pdei ;

    // get the dde pointer

```

```

pddei = (PDDEINIT)mp2 ;

// store the client HWND
if( !strcmpi( pddei -> pszAppName, PMSERVER) && !hwndClient)
    hwndClient = HWNDFROMMP( mp1) ;
...
}
break ;
...

```

It is impossible to predict the outcome of the search. All window procedures that have a case `WM_DDE_INITIATE` branch are all candidate servers. The first obstacle to overcome is checking the contents of the `DDEINIT` structure about the application and topic requested. The code of the potential server must therefore look carefully at the members `pszAppName` and `pszTopic` in order to be certain that it is the application being sought, and that it is capable of supporting the required conversation topic. Only if both conditions are met will the potential server notify the client through the message `WM_DDE_INITIATEACK`. The operation is performed by the function *WinDdeRespond()*:

```

#define INCL_WINDDE
MRESULT APIENTRY WinDdeRespond(  HWND hwndClient,
                                HWND hwndServer,
                                PSZ pszAppName,
                                PSZ pszTopicName,
                                PCONVCONTEXT pcctxt) ;

```

<i>Parameter</i>	<i>Description</i>
<code>hwndClient</code>	Handle of the window that will act as the client in the conversation
<code>hwndServer</code>	Handle of the window that will act as the server in the conversation
<code>pszAppName</code>	Name of the application being sought for as the server
<code>pszTopicName</code>	Name of the desired conversation topic
<code>pcctxt</code>	Address of a <code>CONVCONTEXT</code> structure
<i>Return Value</i>	<i>Description</i>
<code>BOOL</code>	Success or failure of the operation

The required information includes the handle of the receiving window as well as that of the sender, in addition to the text strings described previously for *WinDdeInitiate()*.

During this phase of the settlement of a DDE conversation, many different scenarios are possible. If the client seeks a specific server and a specific topic, then the values in `pszAppName` and `pszTopicName` must be precisely what was passed by the client through *WinDdeInitiate()*. At the opposite extreme, it is possible to hypothesize that the client is willing to converse with any other application about any topic; then both text strings were originally set to `NULL`. The server being contacted notifies the client

about which topics it is “proficient” in by generating a number of calls to *WinDdeRespond()* equal to the number of known topics. In the first case, the client receives the WM\_DDE\_INITIATEACK message only. In the second case, it receives as many messages as there were calls to *WinDdeRespond()*. According to the DDE specifications, the client must handle each WM\_DDE\_INITIATEACK by responding with a WM\_DDE\_TERMINATE for all conversation topics which are irrelevant.

```

...
case WM_DDE_INITIATE:
{
    PDDEINIT pddei ;
    CONVCONTEXT conv ;

    conv.cb = sizeof( CONVCONTEXT ) ;
    conv.fsContext = DDECTXT_CASESENSITIVE ;
    conv.idCountry = 39 ;
    conv.usCodepage = 437 ;
    conv.usLangID = 0 ;
    conv.usSubLangID = 0 ;

    // skip if we are already connected
    if( hwndClient)
        break ;

    // get the dde pointer
    pddei = (PDDEINIT)mp2 ;

    // store the client HWND
    if( !strcmpi( pddei -> pszAppName, PMSERVER) && !hwndClient)
        hwndClient = HWNDFROMMP( mp1) ;

    // start the conversation
    WinDdeRespond( hwndClient, hwnd, PMCLIENT,
                  pddei -> pszTopic, &conv) ;
}
break ;
...

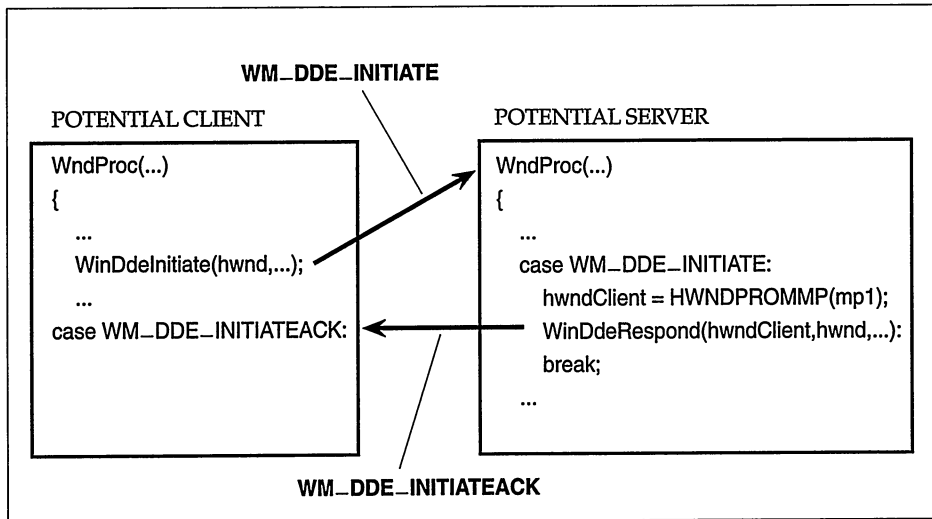
```

For the sake of simplicity, let’s examine the case of a conversation between a client and a predetermined server about a specific topic known to both of them. Only when the client receives the message WM\_DDE\_INITIATEACK is it possible to state with certainty that the two applications are engaged in a DDE conversation. Figure 11.11 illustrates the relationship between the two applications during this phase of the establishment of a DDE conversation.

## Requesting Data

After the client window has settled a DDE conversation, it can ask its server to provide data. This might happen immediately or minutes later. The preliminary actions between the client and the server have been rendered through the functions *WinDdeInitiate()* and





**Figure 11.11** Definition of the relationship between a client and its server in a DDE conversation.

*WinDdeRespond()*, which produce as a side effect *broadcasting* and *sending* the messages `WM_DDE_INITIATE` and `WM_DDE_INITIATEACK`, respectively. For all ensuing operations of a DDE conversation, you must use the function *WinDdePostMsg()*. As its name suggests, the function *WinDdePostMsg()* will post a `WM_DDE_` message into the target application's queue rather than directly accessing its window procedure.

```

#define INCL_WINDDE
BOOL APIENTRY WinDdePostMsg( HWND hwndTo,
                             HWND hwndFrom,
                             ULONG wm,
                             PDDESTRUCT pdest,
                             ULONG flOptions) ;
  
```

<i>Parameter</i>	<i>Description</i>
<code>hwndTo</code>	Handle of the window receiving the message
<code>hwndFrom</code>	Handle of the window sending the message
<code>wm</code>	DDE message to be conveyed
<code>pdest</code>	Address of the passed data structure
<code>flOptions</code>	Options pertaining to the repetition of the insertion of the message into the target application's queue ( <code>DDEPM_RETRY</code> ), and to the rules dealing with the disposal of memory ( <code>DDEPM_NOFREE</code> )
<i>Return Value</i>	<i>Description</i>
<code>BOOL</code>	Success or failure of the operation

As for *WinDdeRespond()*, you must indicate both the handle of the target window as well as that of the issuing window, in addition to the appropriate WM\_DDE\_ message. This ensures that either a client or a server engaged in multiple conversations at the same time can always distinguish them. *WinPostDdeMsg()* is used to transport any of the last eight messages listed in Table 11.3. The fourth parameter is a pointer to a DDESTRUCT structure, which is needed to solve some problems in information passing between the client and the server.

```
typedef struct _DDESTRUCT
{
    ULONG cbData ;
    USHORT fsStatus ;
    USHORT usFormat ;
    USHORT offszItemName ;
    USHORT offabData ;
} DDESTRUCT ;
```

The members of DDESTRUCT are used mainly to describe the variable portion of the data area shared between the client and the server, rather than containing transaction-specific information. Table 11.6 lists the values that can be assigned to the fsStatus member of DDESTRUCT.

For some messages, the value of offszItemName and that of offabData must be set to 0 to indicate that no information should be exchanged between the client and the server. In this case, it is sufficient for the target application to receive a WM\_DDE\_ message to understand what has taken place.

**Table 11.6 Values That Can Be Assigned to the fsStatus Member in the DDESTRUCT Structure**

<i>Flag</i>	<i>Value</i>	<i>Description</i>
DDE_FACK	0x0001	Acknowledge response message.
DDE_FBUSY	0x0002	The application is busy.
DDE_FNODATA	0x0004	Data transfer is not to be performed with WM_DDE_DATA.
DDE_FACKREQ	0x0008	Requires that an acknowledge message be issued to confirm an action.
DDE_FRESPONSE	0x0010	Response to a request message.
DDE_NOTPROCESSED	0x0020	The message is not supported.
DDE_FRESERVED	0x00C0	Reserved: must be zero.
DDE_FAPPSTATUS	0xFF00	The eight high bits are reserved for application specific data.

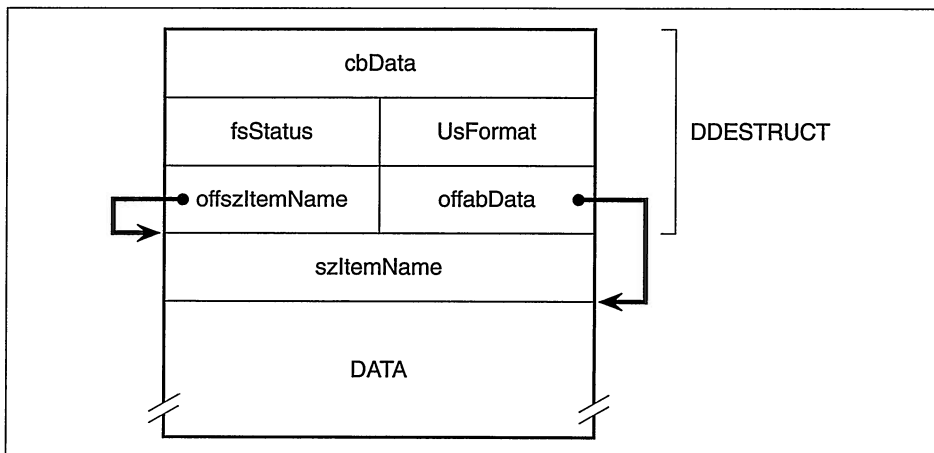
The allocation of a memory block containing the DDESTRUCT structure and the actual data is handled by the application. Naturally, this must be a shared memory area, in order to allow the receiver to dereference any appropriate pointer. The receiver of a posted WM\_DDE\_ message must take care of the disposal of the data area passed to it by calling *DosFreeMem()*. The application that allocates the memory block will get hold of the PID of the receiving process by calling *WinQueryWindowProcess()*, described in the following paragraph. Figure 11.12 illustrates the structure of the memory block associated with the DDESTRUCT structure.

A client process issues a WM\_DDE\_REQUEST message to request some data. The DDESTRUCT contains the name of the item it is requesting, in addition to the transfer format (generally DDEFMT\_TEXT). In addition to the native formats, a process might also define customized formats, although they must be registered in the system through the global atom table so that they become accessible to all applications present in the system. Finally, the client posts the message WM\_DDE\_REQUEST and the shared block to all servers by calling *WinDdePostMsg()*. Then it frees the memory block with *DosFreeMem()*. The samples in Listings 11.3 and 11.4, which you will find on the disk that comes with this book, illustrate in detail all the steps necessary to implement the DDE protocol.



## Providing Data to the Client

Once the request WM\_DDE\_REQUEST has been received, the server sends the requested data to the client, or it responds with a denial message if it is not able to satisfy the request. In both cases, the server follows the approach described for the creation of the DDESTRUCT block. If the server is able to fulfill the request in the specified format, it places the DDESTRUCT structure and its data in a shared block. It then indicates it in a WM\_DDE\_DATA message posted to the client through *WinDdePostMsg()*.



**Figure 11.12** The memory block used for passing information between the server and the client.

```

...
case WM_DDE_REQUEST:
{
    PSZ p, pszString ;
    ULONG ulSize ;
    PDDESTRUCT pddes ;

    // retrieve the stock names from the resource file
    if( DosGetResource( NULLHANDLE, RS_STOCKS, ID_STOCKS,
                      (PVOID)&pszString)
        WinAlarm( HWND_DESKTOP, WA_ERROR) ;

    // get the size
    ulSize = strlen( pszString) ;

    // extract the pddes pointer
    pddes = (PDDESTRUCT)mp2 ;

    // extract the pointer to the item name
    p = DDES_PSZITEMNAME( pddes) ;

    // check if we have requested the handle to the atom table
    if( strcmpi( "stocks", p) == 0 && pddes ->
        usFormat == DDEFMT_TEXT)
    {
        // prepare the memory block to be sent to the client
        pddes = AllocDdeBlock( hwndClient, "stocks",
                              0, DDEFMT_TEXT,
                              pszString, ulSize) ;

        // post the handle to the client
        WinDdePostMsg( hwndClient, hwnd, WM_DDE_DATA, pddes, TRUE) ;

        DosFreeMem( pddes) ;
        DosFreeResource( pszString) ;
    }
}
break ;
...

```

When the client receives the message, it processes the data and frees the memory block. A denial message from the server simply means it is not able to fulfill the WM\_DDE\_REQUEST. For this purpose, it sets the DDE\_NOTPROCESSED status bit in the fsStatus member of the DDESTRUCT structure, and it will post a WM\_DDE\_ACK to the client by means of *WinDdePostMsg()*.

When the server application is busy, it sets the DDE\_FBUSY bit. If the server needs the client to confirm the receipt of the message (whatever type of response is produced by the server), it must also set the DDE\_FACKREQ status bit.

A client receiving a negative response from the server repeats the request for that data using a different format. Generally, the first request is in the most complex format, while further requests will employ simpler and simpler formats.

## *Establishing a Permanent Link*

A client often establishes a permanent link with the server with reference to a specific data item. After setting up the relationship, the server notifies the client that the data has changed, or it will transmit them directly. The data link remains active until it is not explicitly cleared.

To set up a *permanent link*, the client posts a `WM_DDE_ADVISE` message, with the `DDESTRUCT` structure containing the name of the data set. The permanent link is established at two levels: notification from the server of the change of value of the data or actual transfer of the changed values to the client. In order to receive a notification about changes in data, the client must set the `DDE_FNO` bit when it sends the `WM_DDE_ADVISE` message. A server able to fulfill the request (it can access the data and use the required format) stores it in an internal table, and posts an acknowledge message to the client (a `WM_DDE_ACK` with the status bit `DDE_NOTPROCESSED` cleared, and the bit `DDE_FRESPONSE` set). From this moment, any time that a change occurs in the data the server posts a `WM_DDE_DATA` message with the `DDE_FNO` bit set. The client can ignore the notification or request an updated copy of the data through an ordinary transaction based on `WM_DDE_REQUEST`.

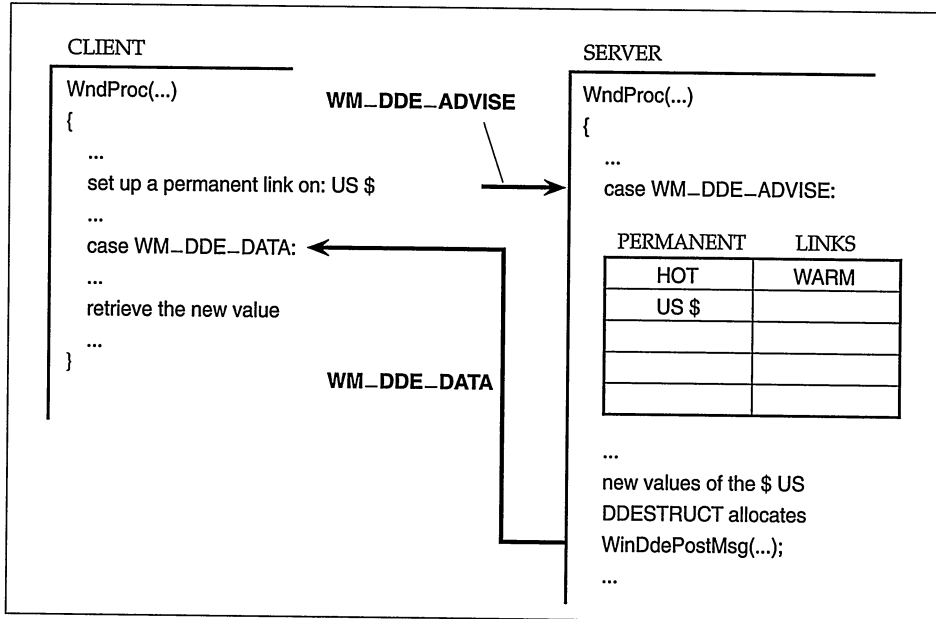
The second kind of permanent link involves the actual transfer of the changed data, and is set up by the client through the `WM_DDE_ADVISE`, without setting the `DDE_FNO` bit in the `DDESTRUCT` structure. The server is obliged to include the updated data in any ensuing `WM_DDE_DATA` message. In both cases, if the server is not able to fulfill the request expressed with `WM_DDE_ADVISE`, then it must respond with a negative message.

The server sets the `DDE_FACK` bit if the client is to return either a positive or negative message after receiving the data. This approach can actually impact on both applications performance, especially if the time interval between any two successive changes to the data set involved in the permanent link is short. However, this constant exchange of messages also grants a greater level of security in handling the data.

To terminate a link established through `WM_DDE_ADVISE`, the client posts a `WM_DDE_UNADVISE`. Then it is up to the server to figure out which clients still have active permanent links by examining the internal table and removing the appropriate entry from it (Figure 11.13).

## *Terminating a DDE Conversation*

A window can terminate a DDE conversation at any time by posting a `WM_DDE_TERMINATE`. This is the only DDE transaction that can be generated indifferently by both the client or the server. The message `WM_DDE_TERMINATE` immediately breaks off all transactions in a conversation. The window should not post any other message in that



**Figure 11.13** Passing data to the client.

conversation. However, there is an exception: When the target receives a `WM_DDE_TERMINATE`, it must immediately post the same message. Consequently, the window that initiated the termination sequence for the conversation with a `WM_DDE_TERMINATE` must be ready to receive the same message from its partner. Only then is it possible to destroy the window involved in the DDE. Figure 11.14 illustrates the termination sequence of a DDE conversation.

## *Invisible Windows*

The most interesting aspect in the implementation of the DDE communication protocol is using invisible windows, children to `HWND_OBJECT`, within a conversation. Although the message handling in a DDE conversation might be handled in the window procedure of the main window, this solution produces a rather complicated window procedure, especially because the same program can play different roles (both client and server), or be engaged in several conversations at the same time. In some cases, though, the situation will break. The best approach is to create two new kinds of classes (window procedures) for processing the message flow pertaining to the roles of client and server in the DDE conversation. To make this solution even more efficient, it is best to assign to each new window type a message queue of their own, and thereby place them in distinct threads. The situation is illustrated in Figure 11.15.

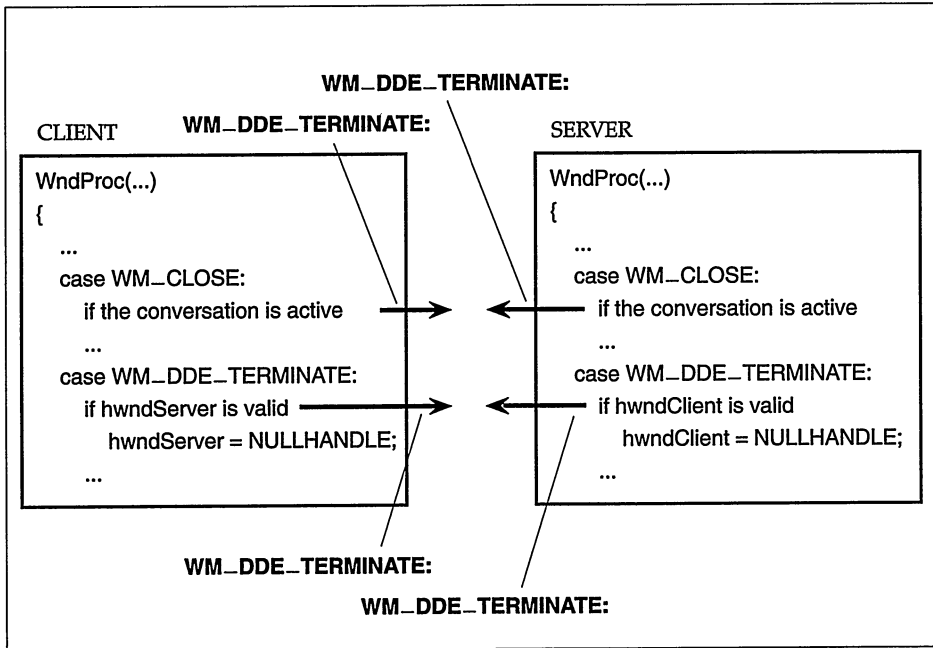


Figure 11.14 Termination sequence of a DDE conversation.

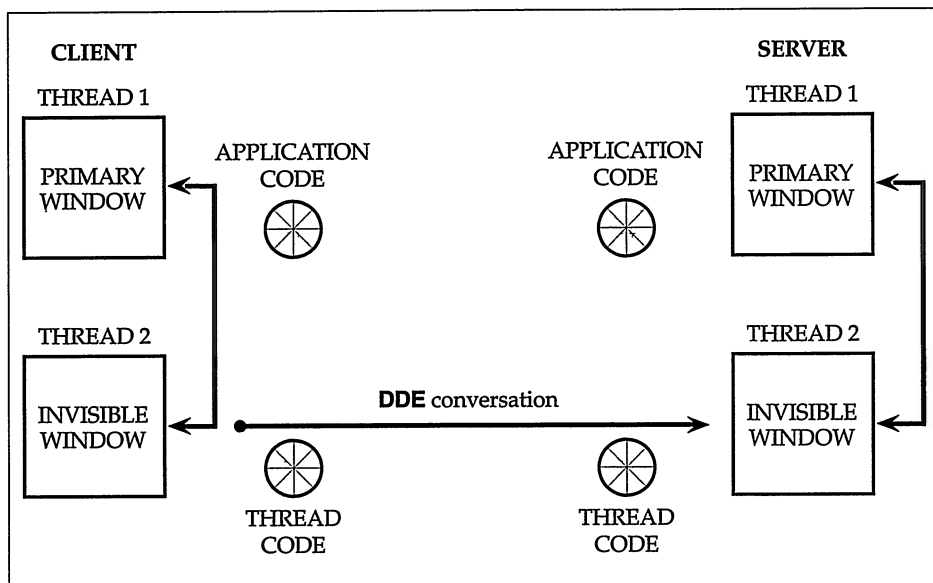


Figure 11.15 Handling a DDE conversation through an invisible window.

## Uses of DDE

A typical example of a DDE conversation is that of a server application that retrieves data about stock quotations and passes them to another application that will further process it.

In Listings 11.3 and 11.4 we will pretend that a PM server accesses some theoretical financial data and sends it periodically to a requesting client. Writing two applications with well-defined roles, as in this case, be careful with the client's "look and feel." The server can operate like an icon or even invisibly; it is its internal logic that is interesting, not its external look.

## Defining the Project

To illustrate the DDE communication protocol, we implement an application capable of accessing a server named PMSERVER which supports three topics. Let's call them NASDAQ, AMEX, and NYSE, corresponding to the names of the three major stock exchanges in New York. The first operation of the client, CLIENT.EXE, is selecting the desired server. The CLIENT application will have a top level menu, Market (see Figure 11.16), containing three menu items for selecting the intended financial market.

The execution of this operation involves the receipt of the message WM\_COMMAND in the window procedure of the application's client window. The ID MN\_NASDAQ corresponds to this menu item in the appropriate menu template in the resource file. The first operation is seeking a server with the name NASDAQ by calling *WinDdeInitiate()*. This function will broadcast the message WM\_DDE\_INITIATE to all top level windows present in PM and will return only after having explored all window procedures. The entire operation is synchronous in nature. We might safely expect that the presence of the NASDAQ topic will respond with the message WM\_DDE\_INITIATEACK to *WinDdeInitiate()*. When the flow of execution returns from this function, the client application knows if there is a server capable of satisfying its needs.

The action taken by the SERVER application when it receives the message WM\_DDE\_INITIATE is very simple. The program checks that the name of the required server matches its own name, and then returns to CLIENT the same data it received, by means of a WM\_DDE\_INITIATEACK passed with *WinDdeRespond()*. The name of the server application being sought is always PMSERVER, while the topic of conversation can be NASDAQ, AMEX, or NYSE, according to the user's choice.

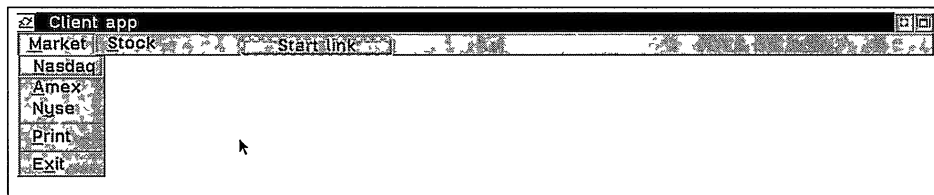


Figure 11.16 CLIENT allows you to select a server providing financial data from NASDAQ, NYSE, and AMEX.



In designing these applications, we let the server notify the client about the whole set of shares used by the client to perform either a search or an analysis. Often, this phase takes place directly when the message WM\_DDE\_INITIATE is received, and forces the server to respond with as many WM\_DDE\_INITIATEACK messages as there are supported conversation topics. In this approach to the problem, we prefer to follow a slightly different route. As you can see in Figure 11.17, the conversation topic is always the name of the stock market, while the single stock titles are the conversation items.

This choice will make it easier to implement the termination condition of the DDE conversation. According to the DDE specifications, in response to each WM\_DDE\_INITIATEACK issued by the server to notify the client about the supported conversation topics, the client should issue a WM\_DDE\_TERMINATE for each irrelevant topic, thus indicating the stock title covered. The definition of the conversation topic as the NASDAQ stock exchange allows us to pass to the client the set of stocks supported by the server directly when the conversation is established. An additional advantage of this approach is a considerable memory saving. During the initial activation phase of a DDE conversation, it is up to PM to define and free any memory area shared among different applications by calling *DosAllocSharedMem()* with the flag OBJ\_GIVEABLE. Maintaining a number of open conversations could cause some problems or performance degradation, especially if the system is short of memory.

Once the initial phase of the DDE conversation is over, the client has an exhaustive list of all stock titles supported by the server. Then it is up to the user to determine which shares to use. In CLIENT, the menu item Query NASDAQ will be active (Figure 11.18) so as to allow the user to make his selection.

The selection of this option involves the appearance of a dialog window containing in a listbox the names of the listed shares currently supported by the server. The user selects the one to examine. The outcome is the formation of a link between the client and the server on the selected topic. Then, the user has two alternatives: request the current exchange quotations of the stock on a continuous basis (*permanent link*) or just when it is deemed necessary (*one-time link*).

The first solution is aesthetically more appealing, since it will make a listbox appear in the left portion of the program's client window, where the data passed from the server is continuously updated (Figure 11.19).

The transfer of data will go on for an unlimited time, even if it would be useful to perform some statistical analysis on the information received (Figure 11.20). That's why there is a Statistics menu item in the Query top level. This will compute the mean,

Application	PMSERVER
Topic	NASDAQ
Item	AST, MSFT, ...

**Figure 11.17** The DDE conversation items between CLIENT and SERVER.

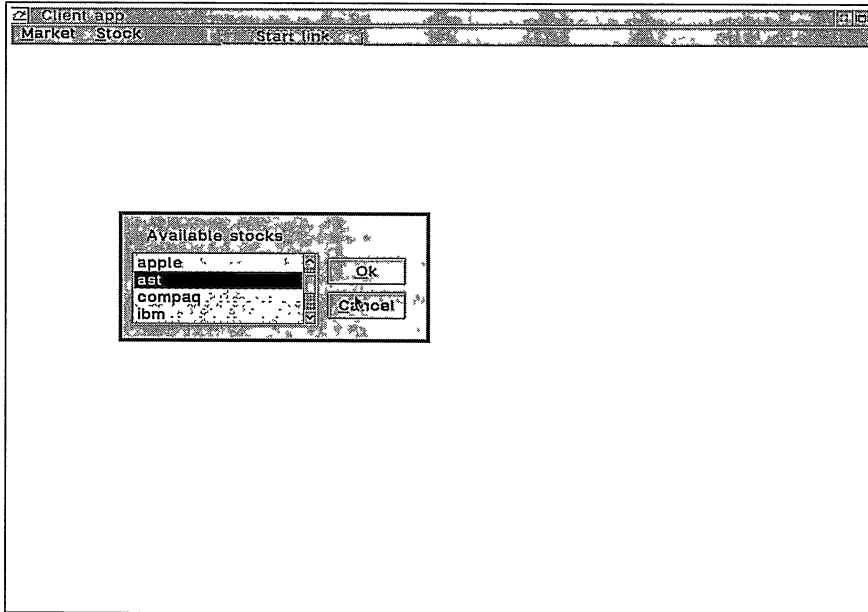


Figure 11.18 Activation of the menu Query NASDAQ after selecting the corresponding stock market.

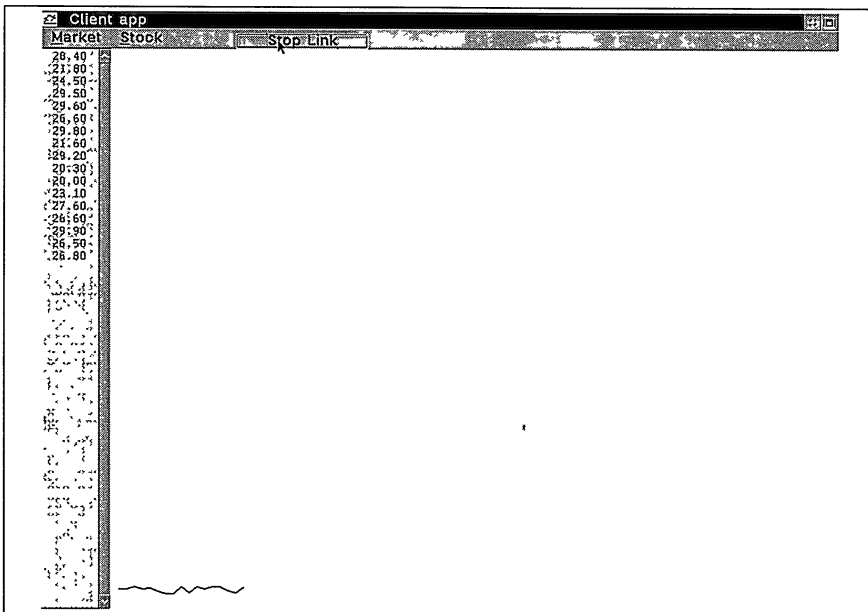
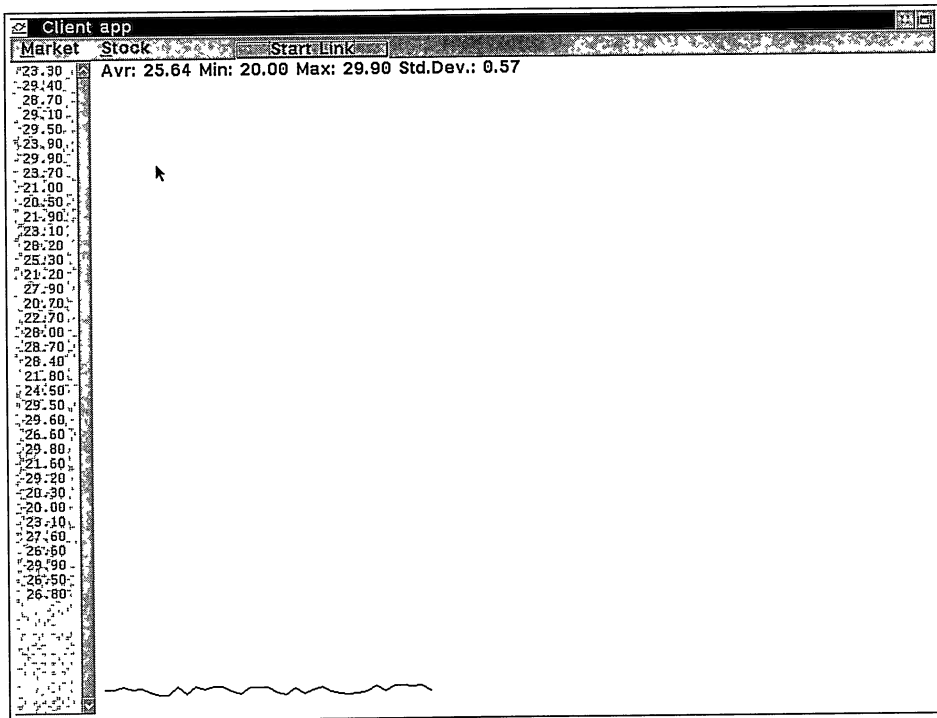


Figure 11.19 CLIENT in action after permanent link has been activated for displaying the ongoing changes to a stock quotation.



**Figure 11.20** Presentation of a set of statistical values on the data received by CLIENT from SERVER.

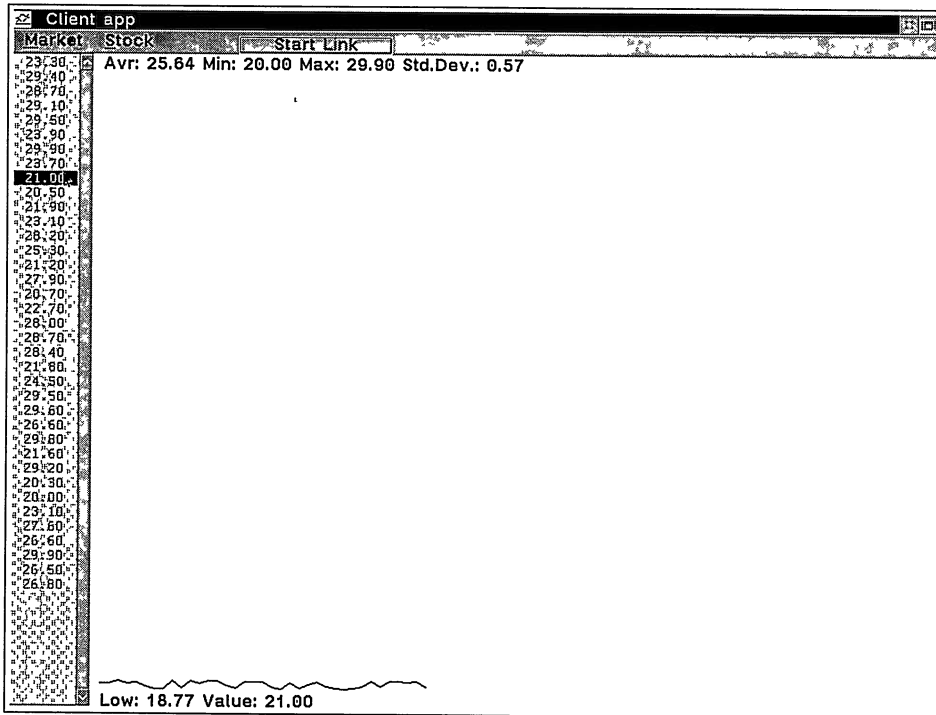
the standard deviation, the minimum, and the maximum value, and present these values in a window to the side of the menu bar.

At the very moment when the data is received, the user can get some additional information, with respect to what is displayed numerically or graphically by the application. A double click on any item in the list box will cause another window of facts to appear in the lower part of the screen; this window will indicate, in addition to the selected value, its minimum and maximum during the day (Figure 11.21).

## The Server

The duty of the server is to satisfy the requests of the client who is querying it. In general, a DDE server must be extremely flexible in supporting a high number of transactions in different formats. In the specific case of the SERVER application, the interaction with the CLIENT is basically limited to two operations: passing the names of all stocks supported when the conversation is established, and then sending data about a stock item after a *permanent link* has been set up by CLIENT.

Naturally, since in this example it is not possible to access a real source of information, the sample server application will invent both the stock names as well as their



**Figure 11.21** A double-click on any listbox item will display another information window.

values. The complete list of available stocks for each server is limited to a few, all taken from the *Wall Street Journal Europe*. The simplest solution is a traditional set of strings inserted in the `STRINGTABLE`, then loaded with `WinLoadString()`. Instead of following this standard approach, here we use the function `DosGetResource()`, which should consume less memory and perform faster. With `DosGetResource()` it is possible to access a block of memory in read-only mode containing the specific resource file. The ASCII file used to list the supported stock titles (`STOCKS.DOC`) looks like this:

```
ast,ibm,apple,msft,compaq,aldus,intel,lotus,
```

The file `VALUES.DOC` is used by the server to define the range of fluctuation of each title. These values correspond to the minimum and maximum registered during the last 52 weeks of dealing.

```
20-30,90-140,50-60,90-120,30-70,40-50,40-60,16-40,
```

The portion of the resource file involved in this operation is the following one:

```
...
RESOURCE RS_STOCKS ID_STOCKS stocks.doc
RESOURCE RS_VALUES ID_VALUES values.doc
...
```

This operation implies the allocation of a block of shared memory passed from the server to the client (it would be impossible to act from the client through a pointer provided by *DosGetResource()* because the operation would affect a memory area not belonging to the application). Therefore, it is the client's responsibility to read the stock names supported by the server as soon as it receives the message WM\_DDE\_DATA.

The establishment of a *permanent link* between the client and the server requires that the message WM\_DDE\_ADVISE be issued. In this case, the client notifies the server about the stock name it is interested in by passing a numerical ID (converted to a string) rather than the actual stock name because the server, once it has completed the tasks involved with the initial WM\_DDE\_REQUEST message from the client (immediately after the establishment of the conversation), gets rid of the resource retrieved through *DosGetResource()*, and will no longer know stock titles it can access.

When passing WM\_DDE\_ messages by means of *WinDdePostMsg()*, it is necessary for the application to allocate a block of memory large enough to contain a DDESTRUCT structure plus some other additional data. The allocation of a data area depends both on the size of the DDESTRUCT structure as well as on the presence of a text string and data items that additionally qualify the nature of the transaction. The memory block is created with *DosAllocSharedMem()* by specifying the flag OBJ\_GIVEABLE. Furthermore, the sender has to be able to get to a valid pointer in the receiver's process. To make the allocated memory block sharable and accessible, it is necessary to know the PID of the receiving process, which is a parameter of the *DosGiveSharedMem()* function. This piece of information can be retrieved in the OS/2 kernel only from within processes that are strictly related—parent and child, for instance—while in PM there is a specific function, *WinQueryWindowProcess()*, that allows the user to find both the PID as well as the TID of the process to which any given window belongs. The syntax of this function is the following:

```
#define INCL_WINMESSAGEMGR
BOOL WinQueryWindowProcess( HWND hwndDest,
                             PPID ppid,
                             PTID Ptid ) ;
```

<i>Parameter</i>	<i>Description</i>
hwndDest	Handle of the window belonging to the process being queried for
ppid	Address of an identifier of the type PID, wherein the process's PID will be written
ptid	Address of an identifier of the type TID, wherein the process's TID will be written
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

Both CLIENT and SERVER refer to the function *AllocDdeBlock()* that allocates a shared memory block appropriate for a DDE conversation. Once the receiving process's PID is available, it is possible to determine the pointer by calling *DosGiveSharedMem()*:

```

...
// determine the client PID
WinQueryWindowProcess( hwndDest, &pid, &tid );
DosGiveSharedMem( (PVOID)pddes, pid, PAG_READ | PAG_WRITE );
...

```

When processing the message WM\_DDE\_ADVISE, the server performs two actions. It retrieves from the resource file the fluctuation range for the involved stock title, and then activates a timer to make sure that a WM\_DDE\_DATA message will be sent to its client at a preset rate (once a second).

The action of reading the contents of the new memory block retrieved with *DosGetResource()* and retrieving the fluctuation range is based on the ID provided by the client. This value is the progressive insertion number of the listbox in the selection listbox in the server. The information is generated in the client when interpreting the block of memory provided by the server, and stored by means of the LM\_SETITEMHANDLE message into the four bytes available for each item in the listbox.

When processing the message WM\_TIMER, the server generates a random float value corresponding to the fictitious daily quotation of the stock title. In addition to this data, a percentage of reduction and increase is also computed in order to define the fluctuation range during the day. All three values constitute the set of data that the server transfers to the client.

## The Client

Interactions come from the server through the messages: WM\_DDE\_INITIATEACK, WM\_DDE\_ACK, WM\_DDE\_DATA, and WM\_DDE\_TERMINATE. Processing WM\_DDE\_INITIATEACK implies the creation of a shared memory area to ask the server the names of available shares. Processing the message WM\_DDE\_DATA involves a test to check the source of the message. This could be the initial response to the request for listing all shares supported by the server, one of the many updates regarding a permanent link, or even the transfer of data asked for through a direct request.

The message WM\_DDE\_ACK reaches the client as the positive reply of the server after a request for the establishment of a permanent link. When the user presses the button Stop Link, the link with the server will automatically be broken, and cause the message WM\_DDE\_UNADVISE to be issued. The statistical computation performed by the client on the data present in the listbox is an excellent example of dynamic interaction with this window. After determining the number of items present, the code can proceed with the computation of the mean and the standard deviation with no performance penalty and with only a limited number of statements.

## The Client Interface

The client process encompasses the totality of activities that pertain to the interaction with the user. Although the terminology employed (client-server) for the DDE communication protocol implies rules and concepts very different from that which happens

in an application developed for a client-server network architecture, the role played by the DDE client is fundamentally similar to that of a front end in a network. Precisely for this reason it is necessary to be very careful with the application's user interface details, in order to make it simple and easy for the user to interact with the application.

As you can see in the previous figures, in CLIENT there will appear a listbox, and if the user requests, two windows of the class WC\_STATIC positioned directly in the client window, while the menu bar contains a pushbutton and a static window. This seeming breaking of the rules of the traditional interface of a PM application actually demonstrates the flexibility of the PM API.

Placing a pushbutton and/or a static window in the menu bar in PM is easy, thanks to the development model underlying the whole environment. The function *WinCreateWindow()* allows the user to specify the owner as well as the parent of a PM window. For the pushbutton and the window that shows the chosen stock market name and the stock title, the parent is the menu bar, while the owner is the application's client window. This solution provides that the program will operate correctly as far as the handling of screen pixels is concerned—as this task is actually delegated to the menu window—and the handling of the messages generated with the pushbutton, which is dealt with by the client window. The only thing to be careful about is that the flag WS\_CLIPCHILDREN must be set in the menu bar. This attribute is declared with the value of 0x2000000 in the file PMWIN.H, and is absent in all ordinary menu bars.

Adding this attribute to those that are already set for this window is very simple. First, you need to retrieve with *WinQueryWindowULong()* the set of style bits assigned by PM when the window was created on the basis of the information present in the menu template in the resource file. The value returned is then added to flag WS\_CLIPCHILDREN, and then *WinSetWindowULong()* is called to assign the new attribute set to the window. You can also use *WinSetWindowBits()*. There is even a simpler solution to this problem: just write WS\_CLIPCHILDREN in the menu template of the CLIENT application!

The choice of acting directly on the contents of the reserved memory of a window has been applied even to the pushbutton present in the menu bar. Pressing a window of the class WC\_BUTTON with the style BS\_PUSHBUTTON will cause the message WM\_COMMAND to be sent to its owner, exactly as with an ordinary menu item. When the pushbutton shows the string Start Link, the value generated by the button pressure is equal to MN\_STARTLINK. Immediately after that, the application changes the ID of this window by calling *WinSetWindowUShort()* and assigning it the define MN\_STOPLINK. The same pushbutton action produces two different values according to the time of the realization.

We will not examine the logic governing the generation of the graph of the title, since the *Gpi* API is used in a very primitive way in this example. However, it is interesting to look at the mechanism used in registering in the listbox the data items received from the server. The information coming from the server is stored in the shared memory block, which is then destroyed by the client as soon as it has been read. The placement of each single data item in the listbox warrants that the data series can

be accessed with a high degree of flexibility. The computation of some statistical values demonstrates this. Furthermore, the four bytes available for each item will store the decrease and increase percentages with respect to fixing the quotation in order to be able to compute a fictitious minimum and maximum value recorded during the day. According to this scheme, these data items are encapsulated with respect to the data to which they refer—the value of the quotation—and occupy only 50 percent of the memory (two shorts instead of two floats).

### *Some Considerations*

The rules to follow in the implementation of a DDE conversation are simple and limited in number. Their implementation is somewhat less simple, since it requires handling and debugging two programs simultaneously. It is important to remember to free the pointer regarding the data segment received by the action performed by *WinDdePostMsg()* in order to avoid overcrowding the memory, and, above all, deadlock situations when operating with IPMD.

Furthermore, it is necessary to discriminate between the implementation of a client and a server in a DDE link, as in the proposed sample, and an application that can interact with third party products both as the client as well as the server. The implementation rules of the DDE protocol can be adapted and manipulated to your own advantage in the first instance, because you are certain to know precisely what set of messages will show up in the conversation.



## Drag & Drop

At this point we can deepen our knowledge about PM programming by exploring some new features of the development model introduced with version 2.x. The first and most evident one is that of operations generally known as *drag & drop*. The possibility of dragging and dropping any object inside WPS by consistently applying the same set of operations is a big advantage for the user. In MS Windows 3.1, selecting and dragging icons can have as their target any open and active window, provided it supports drag & drop events. The source of information is always File Manager. In OS/2 2.x the user can transfer any kind of object from any folder to any other folder, without bothering about the nature of the target. This method will often obviate the need for the user to resort to the options of the menu bar, and makes the PC a lot more intuitive.

The software designer must plan for support for drag & drop actions. One of the advantages of drag & drop is that, once you have learned its implementation rules, it can be applied to a multitude of situations. Furthermore, once you appreciate the advantages of drag & drop, you will be able to extend this way of working to any object present in WPS, and will be perplexed if you aren't able to do so with all your applications.

So, even if you might be skeptical about the subject, you should, however, be aware of its importance. Let's now examine the API involved in the direct manipulation of objects.

---

### The Drag & Drop API

The functions of drag & drop are all introduced by the prefix *Drg*, and add up to as many as 33 functions (OS/2 2.1). Their prototypes are contained in the header file PMSTDDL.G.H, while the API itself resides in the PMDRAG.DLL module. An alphabetical list with a brief description of these functions appears in Table 12.1.

The implementation of drag & drop is not that complex an operation. First of all, you need to identify the "characters" of a drag & drop, where to move them from, and in what position to drop them. This scheme is ideal for the task of transferring a file from one folder to another to fulfill the need of some move or copy operation. A user of an OS/2 2.1 system will perform this kind of activity often, especially when

**Table 12.1 List of Functions Involved in a Drag & Drop Process**

<i>Drag &amp; Drop Functions</i>	<i>Description</i>
DrgAcceptDroppedFiles	Handles the operations involved in receiving a group of files.
DrgAccessDraginfo	Allows access to the memory segment allocated when a drag & drop operation occurs.
DrgAddStrHandle	Returns a handle to a text string.
DrgAllocDraginfo	Allocates a memory area containing the information of the DRAGINFO structure.
DrgAllocDragtransfer	Allocates a memory block for containing structures of the DRAGTRANSFER type.
DrgDeleteDraginfoStrHandles	Deletes the text strings associated with the handles in a DRAGINFO structure.
DrgDeleteStrHandle	Selectively deletes a text string through its handle.
DrgDrag	Activates a drag operation.
DrgDragFiles	Initiates the operations of direct manipulation of one or more files.
DrgFreeDraginfo	Frees the memory associated with a DRAGINFO structure.
DrgFreeDragtransfer	Frees the memory associated with a DRAGTRANSFER structure.
DrgGetPS	Returns a handle to the presentation space of the target window and emphasizes it.
DrgPostTransferMsg	Posts a message to the other window involved in a drag & drop operation.
DrgPushDraginfo	Allows even a process that is not directly involved in a drag & drop operation to access a DRAGINFO structure.
DrgQueryDragitem	Copies the information of a DRAGITEM structure.
DrgQueryDragitemCount	Returns the number of DRAGITEM structures associated with a DRAGINFO structure.
DrgQueryDragitemPtr	Returns a pointer to a DRAGITEM structure.
DrgQueryNativeRMF	Returns the rendering mechanism/format of the dragging.
DrgQueryNativeRMFLen	Returns the length of the rendering mechanism /format of the dragging.
DrgQueryStrName	Returns the text string associated with a handle to a string.
DrgQueryStrNameLen	Returns the length of a string identified by handle.
DrgQueryTrueType	Returns the native type of an object.

*(continued)*

Table 12.1 (Continued)

<i>Drag &amp; Drop Functions</i>	<i>Description</i>
DrgQueryTrueTypeLen	Returns the length of a string relative to the type of an object.
DrgReleasePS	Releases the PS used for emphasizing the target window.
DrgSendTransferMsg	Sends a message to an application involved in a drag & drop operation.
DrgSetDragImage	Sets the image associated with the cursor in the dragging phase.
DrgSetDragitem	Stores the information pertaining to one or more DRAGITEM structures.
DrgSetDragPointer	Changes the cursor's shape during the dragging operation.
DrgVerifyNativeRMF	Verifies the native rendering mechanism/format of an object.
DrgVerifyRMF	Verifies the rendering mechanism/format pair of an object.
DrgVerifyTrueType	Verifies the native type of an object.
DrgVerifyType	Verifies the type of an object.
DrgVerifyTypeSet	Verifies several types simultaneously and returns in a string the possible matching type.

organizing a personal desktop. In this scenario, the action is performed on objects that are present directly in the desktop of WPS, or in some open folders. The programmer does not have to do anything in this phase, because it will be the system itself (WPS) that will take care of all details. However, apart from the actual meaning of the operations performed by the user, it will always be possible to distinguish at least two main characters involved in the drag & drop: the source window and the target window. The first one corresponds to the place where the user selects and drags off the one or more objects; the second is the destination surface of those objects. The source and target window can even be the same; it will be the programmers duty to define the meaning of the operation (not witchcraft!) and the code supporting it. This is exactly what happens in WPS.

Having to deal with two windows (the source and the target) is somewhat similar to the operative scheme of the DDE communication protocol examined in Chapter 11. Actually, as we will see in depth later, even a drag & drop can be used as an interprocess communication tool, especially when the target window belongs to a module that is different than the one holding the source window.

The physical action of a drag & drop is therefore strongly related to the windows of PM, while the writing of code of implementing selection and release is less so. In the case of DDE, for instance, once the conversation has been established, there will be a constant exchange of information between the two processes by sending messages. A drag & drop, instead, can be a stand-alone operation handled in specific ways within a single application; it does not have to involve other programs. One purpose in this could be that of simplifying interactions between the user and the application by extending the drag & drop paradigm to simple and frequent operations. For instance, the creation of a new document could be assigned to some *Create another* option present in the window context menu, or it could be the outcome of dragging the titlebar icon into the application's client window. EPM.EXE, the advanced system editor of OS/2, follows this approach.

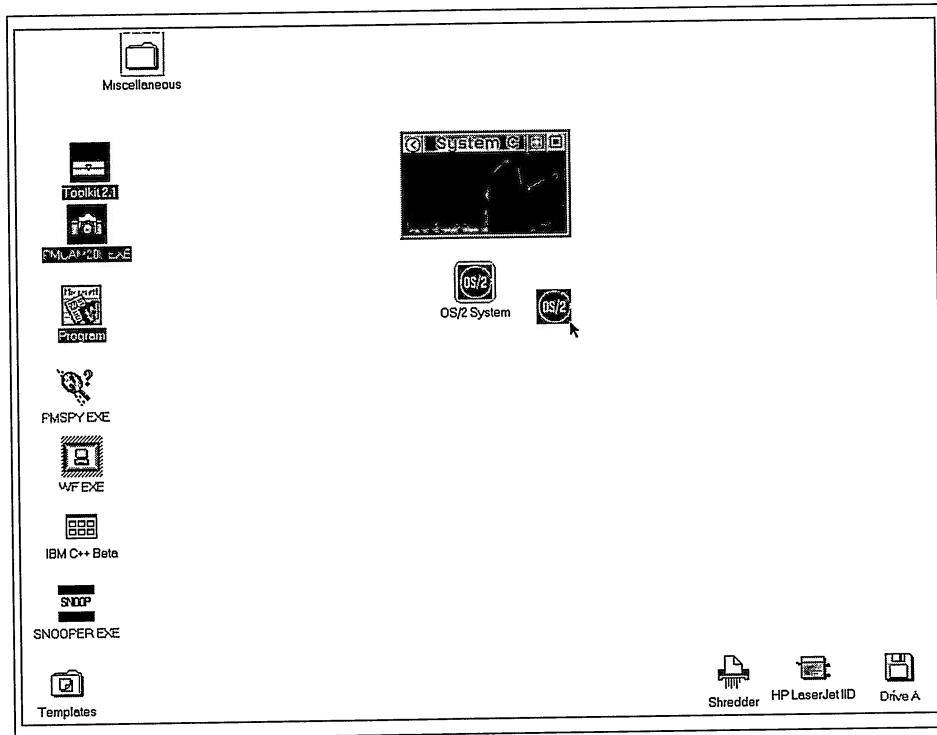
The programmer must cater to the code dealing with the initiation of a *drag* as well as the code accepting a *drop* if that is what the application design demands; then the programmer must define different behaviors according to the level of visual emphasis that will be delivered as feedback to the user.

## The Logic of Drag & Drop

The mouse is the principal tool in executing a drag & drop. By default, the right mouse button (2) will select an object, and possibly drag it, provided the user keeps it depressed. Let's try a quick experiment. In the desktop of WPS there will always be several objects. Select any one of them and keep the right mouse button pressed. What happens? Usually, absolutely nothing. WPS does not change the object's look, and does not display the rounded border to emphasize the object. Try to move the mouse, even just a fraction of an inch, in any direction. Now you can see some changes! The object will be enclosed by a rounded rectangle, and also by a second rectangle (Figure 12.1). Furthermore, the cursor icon will take on a different look. In addition to the slanted arrow pointing northwest, you will see an interdiction sign. All these changes will show the user that a *drag* operation has been initiated.

This is what is perceived by the user. Let's discover what happens in the application at the message flow level. Pressing the right mouse button will cause the message WM\_BUTTON2DOWN to appear in the application's message queue (it is a *posted* message). So far, nothing new. Then the window will receive the message WM\_BUTTON2MOTIONSTART and then WM\_BEGINDRAG. Both messages pass by in the application's queue. When the right mouse button is released, the message pair WM\_BUTTON2MOTIONEND and WM\_ENDDRAG, preceded by a WM\_BUTTON2UP, will appear in the queue. Summarizing, messages appear in the following order:

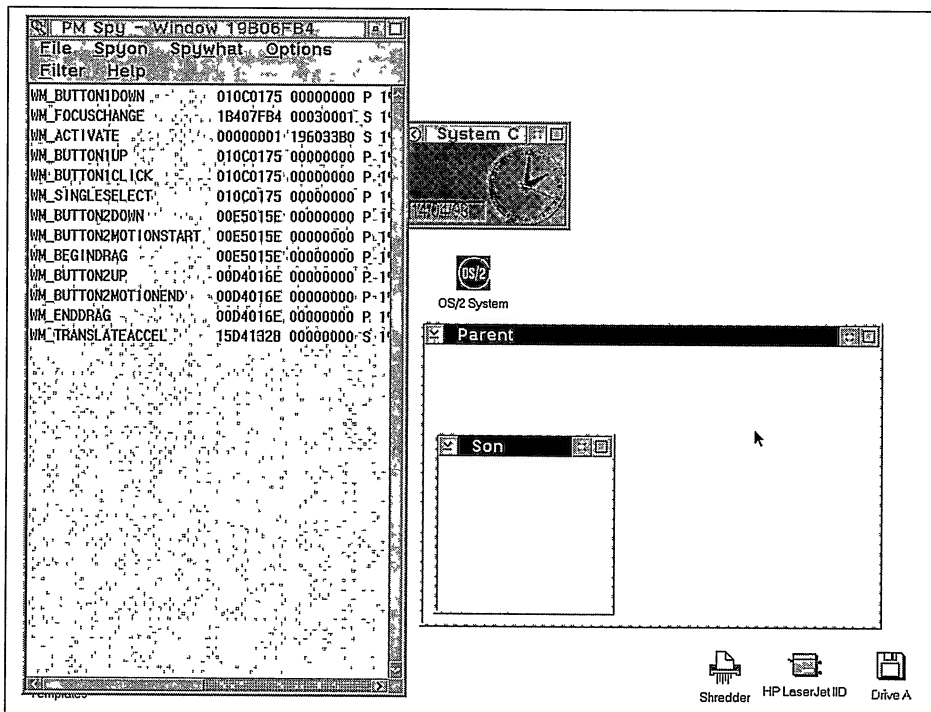
```
WM_BUTTON2DOWN
WM_BUTTON2MOTIONSTART
WM_BEGINDRAG
...
WM_BUTTON2UP
WM_BUTTON2MOTIONEND
WM_ENDDRAG
```



**Figure 12.1** WPS will demonstrate that a drag operation is underway by changing the look of the mouse cursor and enclosing in a rectangle the object holding the hot spot.

The ellipsis indicate the presence (in most situations) of other messages in between the start and the end of the drag & drop. It is interesting to note how the whole thing happens in a natural way as it is handled by the system. To better understand this, try another quick experiment using the example in Listing 4.1, an application that does not utilize drag & drop. Figure 12.2 shows the message flow received by the application's client window after pressing the right mouse button, moving the mouse, and releasing the mouse button.

The first five messages are caused by the transfer of focus and the activation of the application with respect to PMSPY.EXE. Therefore, when you want to activate a drag & drop operation in your own programs, you will only need to prepare appropriate case conditions in the window procedure of the class to which the involved window belongs. In implementing a drag & drop, you do not need to catch the `WM_BUTTON2DOWN` and `WM_BUTTON2UP` messages, since they basically correspond to physical actions performed with the right mouse button. The remaining four messages are truly involved in the implementation of a drag & drop. The syntaxes of `WM_BUTTON2MOTIONSTART` and `WM_BUTTON2MOTIONEND` are identical:



**Figure 12.2** Interception of the message flow in the application's client window simulating the actions involved in a drag & drop operation.

WM_BUTTON2MOTIONSTART	0x0414	<i>Description</i>
mp1	Reserved	
mp2	USHORTfshittestres	Result of the hit-test
Return Value	BOOL fResult	Success or failure
WM_BUTTON2MOTIONEND	0x0415	<i>Description</i>
mp1	Reserved	
mp2	USHORTfshittestres	Result of the hit-test
Return Value	BOOL fResult	Success or failure

Actually, mp1 will contain exactly the same information that you would find in the WM\_BUTTON2DOWN and WM\_BUTTON2UP, which precede and follow in the overall message sequence. In practice, the contents of mp1 and mp2 are almost useless and most often simply ignored. However, it can be productive to intercept these two messages because they correspond exactly with the initiation and termination of a drag & drop operation. You could accommodate in these two case branches code fragments with statements to emphasize even more the start and the end of the operation. There are many choices, from emitting a simple beep to playing back a complete sound arrangement in a

multimedia application. The appearance of the framing rectangle around a WPS object or an object in a folder will happen naturally, and does not require you to write any special code. In other cases, as when dragging the titlebar icon, it is not necessary to adhere to this model, as we will see shortly.

WM_BEGINDRAG	0x0420	<i>Description</i>
mp1	POINTSptspointerpos	Mouse pointer's position in the involved window's coordinate system
mp2	USHORTusPointer	TRUE if the message originates from a mouse action, FALSE if it originates from the keyboard
Return Value	BOOL fResult	Success or failure of the operation
WM_ENDDRAG	0x0421	<i>Description</i>
mp1	POINTSptspointerpos	Mouse pointer's position in the involved window's coordinate system
mp2	USHORTusPointer	TRUE if the message originates from a mouse action, FALSE if it originates from the keyboard
Return Value	BOOL fResult	Success or failure of the operation

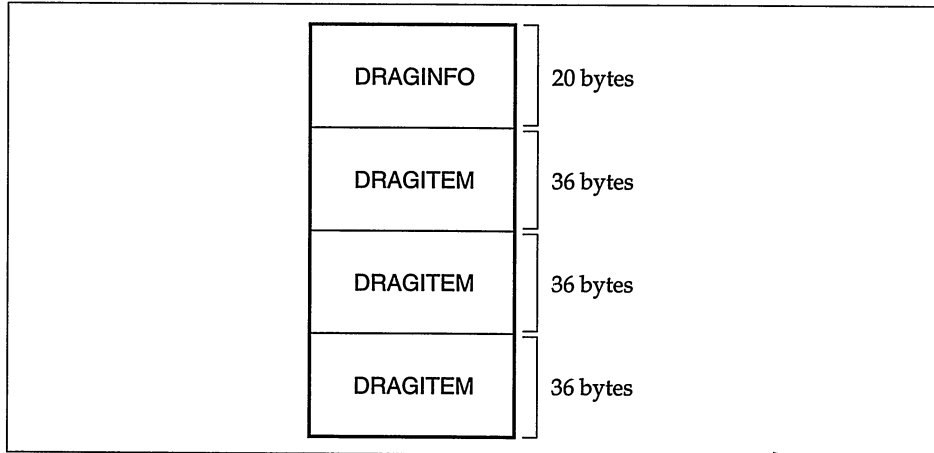
Even in this case, the contents of mp1 and mp2 are not very useful in implementing the code handling a drag & drop operation. In general, knowing the position of the mouse in the window is useful only if there are many different objects, as is the case for a folder (*container*). If, on the other hand, the drag & drop concerns the titlebar icon, or windows containing only one object, you might just as well completely ignore the position of the mouse.

Rather, the message WM\_BEGINDRAG can be exploited to implement all the code necessary to prepare the object associated with the mouse pointer during the drag operation; it is at this time that the specific drag & drop API functions come into play.

## Selecting Objects

The implementation of a drag & drop requires you to allocate a memory block of varying sizes, inside of which you will keep information regarding the selected objects. The memory allocation and handling process is totally delegated to OS/2 by means of the API functions introduced by the *Drg* prefix. The software designer will only have to fill in these areas appropriately, so that even third party applications will be able to access and interpret them correctly. In Figure 12.3 you can see a scheme of the memory area allocated for implementing a drag & drop.

In the code fragment dealing with the WM\_BEGINDRAG message, you have to declare a number of pointers so as to accomplish what is shown in Figure 12.3. The memory block's header corresponds to a DRAGINFO structure, while the information regarding each single object is represented by a DRAGITEM structure.



**Figure 12.3** The initial memory block corresponds to the header, which is then followed by the specific information pertaining to each selected object.

```
typedef struct _DRAGINFO
{ // dinfo
  ULONG cbDraginfo ;
  USHORT cbDragitem ;
  USHORT usOperation ;
  HWND hwndSource ;
  SHORT xDrop ;
  SHORT yDrop ;
  USHORT cditem ;
  USHORT usReserved ;
} DRAGINFO ;

typedef DRAGINFO *PDRAGINFO ;
```

A DRAGINFO structure takes up 20 bytes. Each DRAGITEM will need 36 bytes. On the basis of the selections performed by the user, the application will have to allocate a memory area that is large enough to contain a DRAGINFO structure and a number of DRAGITEM structures matching the objects selected. The operation is not performed explicitly by calling *DosAllocMem()*; it will require you to call *DrgAllocDrginfo()*.

```
#define INCL_WINSTDDRAG
PDRAGINFO APIENTRY DrgAllocDraginfo( ULONG cditem ) ;
```

<i>Parameter</i>	<i>Description</i>
cditem	Number of DRAGITEM structures
<i>Return Value</i>	<i>Description</i>
PDRAGINFO	Pointer to a DRAGINFO structure

The function *DrgAllocDrginfo()* requires as its sole parameter the number of DRAGITEM structures that will follow the DRAGINFO structure—the number of objects



that will be dragged by the user on the basis of the current selection. The return value is a pointer to a memory area beginning with a DRAGINFO structure. Before calling *DrgAllocDraginfo()*, it is necessary to know how many objects have been selected by the user, so that the allocated memory area will be large enough to contain all the information. The members of the DRAGINFO structure will contain data that are automatically assigned by *DrgAllocDraginfo()* and only partially changeable by the application. In the case of single-object dragging, the overall size of the allocated memory block is equal to 56 bytes—20 for the DRAGINFO and 36 for the DRAGITEM. The member `cbDraginfo` will take on the value of 56, while the second one, `cbDragitem`, will be 36. The information of the DRAGINFO end of DRAGITEM is packed one next to the other. Once you have obtained a pointer to the starting memory block, you will be able to find the starting position of all remaining sub-blocks regarding the single DRAGITEM structures. The commencement of a drag & drop inside the WM\_BEGINDRAG message handling code will look like this:

```

...
case WM_BEGINDRAG:
{
    PDRAGINFO pdrginfo ;

    ...
    pdrginfo = DrgAllocDraginfo( 1L) ;
    ...
}
break ;
...

```

The member `usOperation` will contain the value `DO_DEFAULT`, but it can be changed into any of those listed in Table 12.2.

Dragging a document previously produced by a *template* data file is perceived by the target window as a `DO_DEFAULT`. The flag `DO_CREATE`, instead, is the outcome of the dragging of a *template*, an operation that corresponds to the creation of a new instance of that object. The examination of the `DO_` flag becomes vital when writing code dealing with the acceptance of an object, as we will see later. When the information in the DRAGINFO structure is being created, you can change what is defined by the *DrgAllocDraginfo()* function by setting the `usOperation` with the appropriate flag. To indicate that you want to move the selected object, for instance, you need to set the `DO_MOVE` flag:

```

...
pdrginfo = DrgAllocDraginfo( 1L) ;
// it is a move
pdrginfo -> usOperation = DO_MOVE ;
...

```

It is often convenient to also specify the handle of the source window, thereby overwriting the default data:

**Table 12.2 Values of the Member `usOperation` in the `DRAGINFO` Structure**

<i>Drag Operation</i>	<i>Value</i>	<i>Description</i>
DO_DEFAULT	0xBFFE	Default operation.
DO_UNKNOWNN	0xBFFF	Operation defined by the application initiated by pressing nonstandard keys.
DO_COPY	0x0010	Copy operation, the CTRL key is depressed.
DO_MOVE	0x0020	Move operation.
DO_LINK	0x0018	Linking operation, the CTRL+SHIFT key combination is depressed.
DO_CREATE	0x0040	Creation.

```

...
pdrginfo = DrgAllocDraginfo( 1L) ;
// it is a move
pdrginfo -> usOperation = DO_MOVE ;
// source window
pdrginfo -> hwndSource = hwndClient ;
...

```

The choices you make when defining the standard operation with respect to that particular drag & drop operation will affect the visual aspect of the dragged icon. If it is a `DO_MOVE` applied on a Data File, for instance, the icon that looks like a dog-eared sheet will appear at its normal resolution. If it was a `DO_COPY`, then the icon would show up in low intensity, a typical feature of the copy operation of a WPS object.

The members `xDrop` and `yDrop` contain the coordinates, expressed in desktop units, of the object's position when the right mouse button is released. These two values are updated automatically by the system during the drag, and are therefore read-only information for the application (window) over which the right mouse button is released. The `cditem` member contains the number of `DRAGITEM` structures involved, i.e., the parameter of the `DrgAllocDraginfo()` function.

Once this first phase is over, you must fill in the members of the `DRAGITEM` structure, then register the information in the memory area allocated by `DrgAllocDraginfo()` for each selected object. The standard solution is to declare an identifier of the `DRAGITEM` type, then call the function `DrgSetDragitem()`.

```

typedef struct _DRAGITEM
{ // ditem
  HWND hwndItem ;
  ULONG ulItemID ;
  HSTR hstrType ;
  HSTR hstrRMF ;
  HSTR hstrContainerName ;
  HSTR hstrSourceName ;

```

```

    HSTR hstrTargetName ;
    SHORT cxOffset ;
    SHORT cyOffset ;
    USHORT fsControl ;
    USHORT fsSupportedOps ;
} DRAGITEM ;

typedef DRAGITEM *PDRAGITEM ;

```

The 11 members of the DRAGITEM structure describe in great detail each object selected by the user. First comes the handle of the window in which the dragging operation is started, then an ID you can assign as you please. This value is especially useful for distinguishing among different objects in the case of a multiple transfer. The next five members are all of the same type: a handle to a text string. This solution allows the user to limit the size of information passed between processes and allocated in the systems shared memory area. To obtain a handle to a string you can use the function *DrgAddStrHandle()* :

```

#define INCL_WINSTDDRAG
HSTR APIENTRY DrgAddStrHandle( PSZ psz ) ;

```

<i>Parameter</i>	<i>Description</i>
psz	One or more of the defines listed in Table 12.3
<i>Return Value</i>	<i>Description</i>
HSTR	Handle of the specified string

The only parameter of *DrgAddStrHandle()* is a text string containing one or more of the defines listed in Table 12.3, separated by commas. The *hstrType* member is used to indicate one or more defines characterized by the prefix DRT and listed in Table 12.3.

If you have a sharp eye, you will notice that the values of the defines in Table 12.3 correspond to the generic files supported by OS/2 and listed in the Type listbox of the Include page of the Settings notebook of any folder. The following code fragment illustrates how to obtain a string handle:

```

...
case WM_BEGINDRAG:
{
    PDRAGINFO pdrginfo ;
    DRAGITEM drgitem ;

    ...
    drgitem.hstrType = DrgAddStrHandle( DRT_TEXT ) ;
    ...
}
break ;
...

```

**Table 12.3 The Defines to Be Specified in the hstrType Member of the DRAGITEM Structure**

<i>Define</i>	<i>Value</i>
DRT_ASM	"Assembler Code"
DRT_BASIC	"BASIC Code"
DRT_BINDATA	"Binary Data"
DRT_BITMAP	"Bitmap"
DRT_C	"C Code"
DRT_COBOL	"COBOL Code"
DRT_DLL	"Dynamic Link Library"
DRT_DOSCMD	"DOS Command File"
DRT_EXE	"Executable"
DRT_FORTRAN	"FORTRAN Code"
DRT_ICON	"Icon"
DRT_LIB	"Library"
DRT_METAFILE	"Metafile"
DRT_OS2CMD	"OS/2 Command File"
DRT_PASCAL	"Pascal Code"
DRT_RESOURCE	"Resource File"
DRT_TEXT	"Plain Text"
DRT_UNKNOWN	"Unknown"

With the member `hstrRMF` you can set the rendering mechanism and format to be used. This means defining how information will be passed to the target window. Even in this case it is a text string that can contain different mechanism/format pairs, the first one of which sets the best possible combination. Table 12.4 lists all rendering mechanisms, while Table 12.5 lists the different formats.

Data is passed as a file (`DRM_OS2FILE`) or through the DDE protocol (`DRM_DDE`). `DRM_PRINT` interacts with the printer, while `DRM_OBJECT` pertains to WPS objects.

In the text string, the mechanism/format pair must be enclosed by a pair of angled brackets, as in the following example:

```
...
drgitem.hstrRMF = DrgAddStrHandle( "<DRM_OS2FILE,DRF_TEXT>" ) ;
...
```

The three remaining string handles refer to the pathname and the name of the object in the source container, and the name it is given once it is dropped into the destination container.

**Table 12.4 Defines for Rendering Mechanisms Employed by Drag & Drop Operations***Define*


---

```

DRM_DDE
DRM_OBJECT
DRM_OS2FILE
DRM_PRINT

```

---

```

...
drgitem.hstrContainerName = DrgAddStrHandle( "C:\\Desktop" ) ;
drgitem.hstrSourceName = DrgAddStrHandle( "tweny.doc" ) ;
drgitem.hstrTargetName = DrgAddStrHandle("F:\\Desktop\\ TWENY.DOC" ) ;
...

```

The next pair of SHORT identify the displacement of the image displayed during the dragging operation, with respect to the cursor's hot-spot. Values different from zero are useful only if you intend to stretch the image. Otherwise it is advisable to generate a compact combination mouse pointer/dragged image to ease the dragging operation.

The `fsControl` member defines the characteristics of the selected objects. It can be one or more flags introduced by the prefix `DC_`, and listed in Table 12.6.

The last member, `fsSupportedOps`, defines which operations are supported by the source object: Table 12.7 summarizes all possible values. When the application examines

**Table 12.5 Defines for Rendering Formats Employed by Drag & Drop Operations***Define*


---

```

DRT_BITMAP
DRT_DIB
DRT_DIF
DRT_DSPBITMAP
DRT_METAFILE
DRT_OEMTEXT
DRT_OWNERDISPLAY
DRT_PRTPICT
DRT_RTF
DRT_SYLK
DRT_TEXT
DRT_TIFF
DRT_UNKNOWN

```

---

**Table 12.6 Values of the fsControl Member of the DRAGITEM Structure**

<i>Option</i>	<i>Value</i>	<i>Description</i>
DC_OPEN	0x0001	The object is open.
DC_REF	0x0002	A reference to another object.
DC_GROUP	0x0004	A group of objects.
DC_CONTAINER	0x0008	A container with other objects.
DC_PREPARE	0x0010	The source must receive the message DM_RENDERPREPARE before it can pass data to the target.
DC_REMOVEABLEMEDIA	0x0020	The object is on a removable media.

the memory area associated with an object being dragged, it will often find the three flags of Table 12.7 added together. This is what happens, for instance, when dragging a *Plain Text* object (a document produced with the system editor).

Once you have filled in all members of the DRAGITEM structure, you need to insert them in the memory area allocated with *DrgAllocDrginfo()*. PM's API provides the *DrgSetDragitem()* function to perform this task:

```
#define INCL_WINSTDDRAG
BOOL APIENTRY DrgSetDragitem( PDRAGINFO pdinfo,
                              PDRAGITEM pditem,
                              ULONG cbBuffer,
                              ULONG iItem) ;
```

<i>Parameter</i>	<i>Description</i>
pdinfo	Pointer to the memory area allocated by <i>DrgAllocDrginfo()</i>
pditem	Address of a DRAGITEM structure
cbBuffer	Size of the DRAGITEM structures
iItem	Number of DRAGITEM structures
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

Summarizing, the code handling the WM\_BEGINDRAG message will first call *DrgAllocDrginfo()*, then it will fill in a number of DRAGITEM structures equal to the number

**Table 12.7 Operations Supported by the Source Object**

<i>Supported Operations</i>	<i>Value</i>	<i>Description</i>
DO_COPYABLE	0x0001	Supports DO_COPY
DO_MOVEABLE	0x0002	Supports DO_MOVE
DO_LINKABLE	0x0004	Supports DO_LINK

of selected objects, and finally it will call *DrgSetDragitem()* to store this data in the memory block that was allocated.

```

...
case WM_BEGINDRAG:
{
    PDRAGINFO pdrginfo ;
    DRAGITEM drgitem ;

    ...
    pdrginfo = DrgAllocDraginfo( 1L) ;
    ...
    // filling in DRAGITEM structures
    drgitem.hwndItem = hwndClient ;
    ...
    if( !DrgSetDragitem( pdrginfo,
                        &drgitem,
                        sizeof( DRAGITEM),
                        1L))
    {
        WinAlarm( HWND_DESKTOP, WA_ERROR)
        ...
    }
}
break ;
...

```

In this example, only one DRAGITEM structure is involved. For more structures, it might be convenient to declare an array of DRAGITEM structures, and then call *DrgSetDragitem()* only once. Actually, the task performed by *DrgSetDragitem()* is that of a simple memory move, from the identifier *drgitem* declared in the WM\_BEGINDRAG message handling code, to the memory area allocated by *DrgAllocDraginfo()*. Have a look at Figure 12.4.

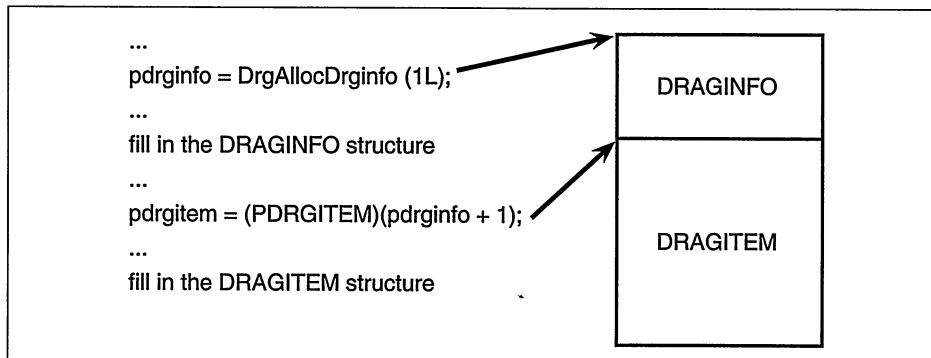
Instead of declaring one or more DRAGITEM identifiers, and then calling *DrgSetDragitem()* to copy their contents, it is possible to obtain a pointer of the PDRAGITEM type, pointing to the appropriate position in the memory block, and then initializing the DRAGITEM directly in the final destination area of all information. The first PDRAGITEM pointer can be obtained in the following way:

```

WM_BEGINDRAG:
{
    PDRAGINFO pdrginfo ;
    PDRAGITEM pdrgitem ;
    ...
    pdrginfo = DrgAllocDraginfo( 3L) ;
    pdrgitem = (PDRGITEM)( pdrginfo + 1) ;
    ...
}

```

while the following ones are the result of a direct increment of the *pdrgitem* pointer.



**Figure 12.4** The constituent parts of the memory block allocated by `DrgAllocDrginfo()`.

## Preparing the Image

An important aspect of drag & drop operations is the definition of the image that will be displayed along with the mouse cursor to make it clear that the operation is taking place. For this purpose, the system provides you with the `DRAGIMAGE` structure:

```

typedef struct _DRAGIMAGE
{ // dimg
  USHORT cb ;
  USHORT cpl ;
  LHANDLE hImage ;
  SIZEL sizlStretch ;
  ULONG fl ;
  SHORT cxOffset ;
  SHORT cyOffset ;
} DRAGIMAGE ;

typedef DRAGIMAGE *PDRAGIMAGE ;

```

The first member contains the size of the structure expressed in bytes. The second one indicates the number of points that make up the image if the field `fl` contains the value `DRG_POLYGON`. If you employ a different flag, the value is set to zero. The third member contains the handle of the image that will be used during the drag operation. This can be a bitmap, an icon, or even a set of points giving rise to a polygonal line. The graphical representation identified by the `hImage` handle can be scaled, and the `SIZEL` member defines its size in the two directions. In general, it is best to use an image in a 1:1 scale factor, especially when dealing with direct manipulation of WPS objects. This is the case, for instance, with folders. If, on the other hand, you're implementing another kind of dragging (like the titlebar icon, or an item from a listbox), it is useful to reduce the image in order to enlarge the user's visual field and help in making precise movements. In the `fl` member there will appear one or more flags that describe the nature of the information specified as the image handle. The choice can be made among the values listed in Table 12.8.



**Table 12.8** Flags and Modifiers Regarding the Image Displayed During Dragging

<i>Flag</i>	<i>Value</i>	<i>Description</i>
DRG_ICON	0x00000001L	Icon.
DRG_BITMAP	0x00000002L	Bitmap.
DRG_POLYGON	0x00000004L	Polygon.
DRG_STRETCH	0x00000008L	The image is stretched.
DRG_TRANSPARENT	0x00000010L	The outline of the icon is drawn.
DRG_CLOSED	0x00000020L	If DRG_CLOSED is set, then a closed polygon is drawn.

The DRAGIMAGE structure is completed by a pair of SHORT that indicate the displacement between the cursor's hot spot and the image displayed during dragging. The number of DRAGIMAGE structures is unrelated to the number of objects actually being dragged (DRAGITEM structures). The relationship can be one-to-one or even one-to-many: one DRAGIMAGE structure for each DRAGITEM, or one DRAGIMAGE for many DRAGITEMs. In between those two extremes, there are several valid combinations.

## Executing the Drag

Once this preparatory phase is over, the application is ready to allow the user to carry out the actual drag. The "magic" of drag & drop is performed by the *DrgDrag()* function. This API function does not return any value until the user releases the right mouse button or presses the ESC key to indicate the end of dragging.

```
#define INCL_WINSTDDRAG
HWND APIENTRY DrgDrag( HWND hwndSource,
                       PDRAGINFO pdinfo,
                       PDRAGIMAGE pdimg,
                       ULONG cdimg,
                       LONG vkTerminate,
                       PVOID pRsvd) ;
```

<i>Parameter</i>	<i>Description</i>
hwndSource	Handle of the source window from which dragging is initiated
pdinfo	Address of a DRAGINFO structure
pdimg	Address of the first DRAGIMAGE structure
cdimg	Number of DRAGIMAGE structures
vkTerminate	Value of the virtual key that must be released to terminate dragging
pRsvd	Reserved
<i>Return Value</i>	<i>Description</i>
HWND	Handle of the window over which the user released the right mouse button

The association of the information indicated in the DRAGIMAGE structure(s) with the other data needed for dragging takes place directly inside *DrgDrag()*. The first parameter corresponds to the handle of the window where the dragging operation is first initiated, followed by the address of the memory block allocated with *DrgAllocDraginfo()*. The handle of the starting window needs not be the same as the actual source of dragging. In a PM window dragging can even originate from the titlebar menu icon (a situation we will examine later). Even in this case, though, it is more convenient to define the starting handle as that of the frame window or of the client window. Since all information is handled internally by the application, there will be no abnormal behavior, and you can choose the window you deem most appropriate.

The pointer to the first DRAGIMAGE structure, and the number of such structures are the next two parameters. The function's syntax is then completed by the virtual key code corresponding to the mouse button that, when released, will signal the end of the dragging; finally, there is a reserved pointer. By default, the termination button will be the mouse's right (2) button and the ESC key. An application can change these values at the system level with *WinSetSysValue()*, or indicate directly in the fifth parameter *DrgDrag()* the define of a different virtual key (the VK\_ defines are listed in PMWIN.H).

For the whole duration of the dragging phase, it is the *DrgDrag()* function that handles all operations with respect to the objects being passed over. More precisely, *DrgDrag()* will keep on seeking a window where it can "deliver" its burden. WPS contains several windows belonging to the WC\_CONTAINER class, and therefore, in this case, you will have the visual feedback that dragging involves objects rather than windows. It is simply a consequence of the nature and implementation of folders.

The last task performed by the source application where all data has been set up for dragging, is freeing the memory area previously allocated by *DrgFreeDraginfo()*. At this point the dragging process is complete.

```
#define INCL_WINSTDDRAG
BOOL APIENTRY DrgFreeDraginfo( PDRAGINFO pdinfo );
```

<i>Parameter</i>	<i>Description</i>
pdinfo	Address of a DRAGINFO structure
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The execution of *DrgFreeDraginfo()* will take place only after the user has released the right mouse button over any window on the screen. The destruction of the allocated data is, in practical terms, the last operation in the dragging process. Let's now examine the other half of the game, that is, dropping.

## Preparing Objects for Dropping

Dragging one or more objects is an operation that is performed completely under the user's control. Inside the system, a thread will be used for executing the *DrgDrag()* that we have seen in the preceding paragraph. The inner workings of this function are very complex, and mostly unknown. However, we can evaluate the results it produces. For instance, during dragging, there will appear some variations in the look of the cursor and of the objects over which it passes displaying an outline border. Sometimes, an interdiction sign will be displayed; at other times, it is the user that changes the look by acting on some keys like CTRL and SHIFT. How does all this happen? Whose responsibility is to make the changes to the windows over which the mouse cursor passes? As you might expect, the job is divided between the *DrgDrag()* and the window over which the mouse's hot spot passes.

The inner workings of *DrgDrag()* will send some messages with the DM\_ prefix directly to the window underlying the mouse's hot spot. The window will catch the DM\_ messages in the window procedure of the class to which it belongs, or it will pass it to the default window procedure. In each case, the underlying window gives the application engaged in the dragging some valuable feedback.

The first area of the drag & drop API that we will explore is that regarding messages (Table 12.9).

The first message received by the window underlying the mouse's hot spot during a dragging is DM\_DRAGOVER. The same window where the selection of objects originated can receive one or more DM\_DRAGOVER messages. It is for this reason that the data structure allocated by *DrgAllocDraginfo()* will, in several instances, contain the handle of the source window. This value is useful in order to prevent a simple *move* operation within the same window. This, however, does not mean that dragging an object in the same window is a senseless operation (just think of making copies or implementing some application-specific functionality that might be different from what is offered by the folders of WPS). Let's have a look at the information conveyed by DM\_DRAGOVER.

		<i>Description</i>
DM_DRAGOVER	0x032E	
mp1	PDRAGINFO pdrginfo	Pointer to the DRAGINFO structure allocated during the set up phase of dragging
mp2	SHORT sxDrop SHORT syDrop	Position of the mouse's cursor (x, y), expressed in desktop coordinates
Return Value	USHORT usDrop  USHORT usDefaultOp	Indicates acceptance/rejection of the objects  Default operation supported by the application

When an application (window) receives the DM\_DRAGOVER message it will be able to:

**Table 12.9 The Messages Involved in Drag & Drop Operations**

<i>Message</i>	<i>Value</i>	<i>Description</i>
DM_DROP	0x032f	Received in the window procedure of the window that formerly responded with DOR_DROP to the message DM_DRAG-OVER.
DM_DRAGOVER	0x032e	Sent to the window underlying the mouse's hot spot.
DM_DRAGLEAVE	0x032d	Sent to the window underlying the mouse's hot spot when the cursor is about to leave the window.
DM_DROPHELP	0x032c	Sent to the window underlying the mouse's hot spot when the user presses the key F1.
DM_ENDCONVERSATION	0x032b	Sent from the target to indicate the termination of a drag & drop operation.
DM_PRINT	0x032a	Sent to execute a print.
DM_RENDER	0x0329	Sent to the source to request a specific object in the predefined rendering format and mechanism.
DM_RENDERCOMPLETE	0x0328	Posted from the source to indicate the termination of <i>rendering</i> operations.
DM_RENDERPREPARE	0x0327	Sent to the source to cause it to set up the <i>rendering</i> of an object.
DM_DRAGFILECOMPLETE	0x0326	Sent at the end of the manipulation of a file in a drag & drop operation.
DM_EMPHASIZETARGET	0x0325	Sent to the target application to indicate the activation or removal of emphasis.
DM_DRAGERROR	0x0324	Sent to signal an error in the execution of <i>DrgDragFiles()</i> or <i>DrgAcceptDroppedFiles()</i> .
DM_FILERENDERED	0x0323	Sent at the end of a file rendering.
DM_RENDERFILE	0x0322	Sent to ask for a file.
DM_DRAGOVERNOTIFY	0x0321	Sent to the source immediately after receiving a message.
DM_PRINTOBJECT	0x0320	Sent to an application supporting the DR-M_PRINT format.
DM_DISCARDOBJECT	0x031f	Sent to a source supporting DRM-DISCARD.

1. Know the nature of what it is being dragged over.
2. Change the look of the cursor appropriately.
3. Activate emphasizing elements of its own surface provided it is able to accept the dragged items.
4. Notify the application (window) that originated the dragging about its consent or denial of the dragged items.
5. Notify the application (window) that originated the dragging about the default operations that it supports.

To implement the first point, it is necessary to transform the value present in `mp1` into a pointer of the `PDRAGINFO` type; but that is not all. The application must call the `DrgAccessDraginfo()` function in order to obtain permission to access that memory area that was allocated in the process that originated the dragging. Behind the scenes of a drag & drop there is a *interprocess communication* (IPC) mechanism that is founded on a shared memory area that is managed through a usage count in order to release it once the operations are terminated.

```
#define INCL_WINSTDDRAG
BOOL APIENTRY DrgAccessDraginfo( PDRAGINFO pdinfo ) ;
```

<i>Parameter</i>	<i>Description</i>
<code>pdinfo</code>	Address of a <code>DRAGINFO</code> structure
<i>Return Value</i>	<i>Description</i>
<code>BOOL</code>	Success or failure of the operation

The parameter of `DrgAccessDraginfo()` is `mp1`, or, as in the following code fragment, a `PDRAGINFO` pointer.

```
...
case DM_DRAGOVER:
{
    PDRAGINFO pdrginfo ;
    ...
    pdrginfo = (PDRAGINFO)mp1 ;
    if( !DrgAccessDraginfo( pdrginfo))
    {
        WinAlarm( HWND_DESKTOP, WA_ERROR ) ;
        ...
    }
    ...
    // decrease the usage counter
    DrgFreeDraginfo( pdrginfo ) ;
    ...
}
...
```

By accessing the information of the dragging's data structure, the receiver is able to understand in an exact way which objects are being dragged over it. The DRAGINFO structure contains three interesting pieces of information:

- The kind of operation supported (USHORT usOperation)
- The handle of the source window (HWND hwndSource)
- The number of associated DRAGITEM structures (USHORT cditem)

Code handling of the DM\_DRAGOVER message involves a switch statement causing the usOperation member to distinguish the various kinds of supported operations (DO\_COPY, DO\_LINK, and the other operations listed in Table 12.2). The handle of the source window can be used to compare it to its own, and then undertake any appropriate action. Finally, to know the number of objects being dragged, you only need to examine the cditem member. Alternatively, PM's API offers the *DrgQueryDragitemCount()* function.

```
#define INCL_WINSTDDRAG
ULONG APIENTRY DrgQueryDragitemCount( PDRAGINFO pdinfo );
```

<i>Parameter</i>	<i>Description</i>
pdinfo	Address of a DRAGINFO structure
<i>Return Value</i>	<i>Description</i>
ULONG	Number of DRAGITEM structures present in the memory area pointed at by the pdinfo pointer

The kind of interaction with the objects is indicated in the return value ensuing the DM\_DRAGOVER message. The response is formulated on the basis of the supported DO\_ operation. In some cases, though, it is necessary to examine more carefully the information at hand to evaluate the contents of any subsequent DRAGITEM structures. The retrieval of a pointer to the first DRAGITEM structure takes place by calling the *DrgQueryDragitemPtr()* function or by acting directly on the PDRAGINFO pointer.

```
#define INCL_WINSTDDRAG
PDRAGITEM APIENTRY DrgQueryDragitemPtr( PDRAGINFO pdinfo, ULONG i );
```

<i>Parameter</i>	<i>Description</i>
pdinfo	Address of a DRAGINFO structure
i	Index of the DRAGITEM structure to be examined, count starts from 0L
<i>Return Value</i>	<i>Description</i>
PDRAGITEM	Address of the i-th DRAGITEM structure

As you might guess, the first parameter of *DrgQueryDragitemPtr()* is the starting address of the memory block (the PDRAGINFO pointer) followed by the index of the DRAGITEM structure, where the first one is identified by the value of 0L. The function returns a pointer of the type PDRAGITEM.

The target window of the DM\_DRAGOVER message now knows about the nature of the object it will receive (*hstrType* member), the kind of proposed *rendering* mechanism and format (*hstrRMF* member), yet more features of the object (*fsControl* member), and the supported operations (*fsSupportedOps*).

The verification of the object type is conducted by retrieving the string and then checking it, or delegating the check to the *DrgVerifyTrueType()* function. The first solution requires a call to *DrgQueryStringName()*, and possibly *DrgQueryStringNameLen()*.

```
#define INCL_WINSTDDRAG
ULONG APIENTRY DrgQueryStrNameLen( HSTR hstr ) ;
```

<i>Parameter</i>	<i>Description</i>
<i>hstr</i>	Handle of a text string
<i>Return Value</i>	<i>Description</i>
ULONG	Length of the string indicated by the handle <i>hstr</i>

This function is rather limited, since the buffers containing text strings pertaining to the type and the rendering mechanism/format will usually be oversized and declared in the program's code as an array of CHAR.

```
#define INCL_WINSTDDRAG
ULONG APIENTRY DrgQueryStringName( HSTR hstr,
                                   ULONG cbBuffer,
                                   PSZ pBuffer ) ;
```

<i>Parameter</i>	<i>Description</i>
<i>hstr</i>	Handle of a text string
<i>cbBuffer</i>	Size of the array of chars declared as the third parameter
<i>pBuffer</i>	Array of chars for containing the text string
<i>Return Value</i>	<i>Description</i>
ULONG	Number of characters actually read

The use of *DrgQueryStringName()* is particularly indicated for examining the members *hstrContainerSource*, *hstrSourceName*, and *hstrTargetName*.

The verification of the type is governed by the three functions *DrgVerifyTrueType()*, *DrgVerifyType()*, and *DrgVerifyTypeSet()*. The first function checks the type of an object (*DRT\_* define) against the first *DRT\_* define present in the string referenced by *hstrType*. With *DrgVerifyType()*, instead, the comparison is for any *DRT\_* define. *DrgVerifyTypeSet()* is used if you need to check for several types at the same time and obtain the text strings corresponding to *DRT\_* actually found.

```
#define INCL_WINSTDDRAG
BOOL APIENTRY DrgVerifyTrueType( PDRAGITEM pditem, PSZ pszType ) ;
```

<i>Parameter</i>	<i>Description</i>
<i>pditem</i>	Pointer to a DRAGITEM structure
<i>pszType</i>	String containing one or more <i>DRT_</i> defines, separated by commas

<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

```
#define INCL_WINSTDDRAG
BOOL APIENTRY DrgVerifyType( PDRAGITEM pditem, PSZ pszType) ;
```

<i>Parameter</i>	<i>Description</i>
pditem	Pointer to a DRAGITEM structure
pszType	String containing one or more DRT_ defines, separated by commas

<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

```
#define INCL_WINSTDDRAG
BOOL APIENTRY DrgVerifyTypeSet( PDRAGITEM pditem,
                                PSZ pszType,
                                ULONG cbMatch,
                                PSZ pszMatch) ;
```

<i>Parameter</i>	<i>Description</i>
pditem	Pointer to a DRAGITEM structure
pszType	String containing one or more DRT_ defines, separated by commas
cbMatch	Size of the array of chars that will contain the string to be checked against
pszMatch	Array of chars containing the string to be checked against

<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The compatibility check is a fundamental test in order to establish the return value generated by processing the DM\_DRAGOVER message.

```
...
case DM_DRAGOVER:
{
    PDRAGITEM pdrgitem ;
    ...
    if( !DrgVerifyTrueType( pdrgitem, DRT_TEXT))
    {
        WinAlarm( HWND_DESKTOP, WA_ERROR) ;
        ...
    }
}
...

```

For hstrRMF it is much more convenient to use *DrgVerifyRMF()* and *DrgVerifyNativeRMF()*.



```
#define INCL_WINSTDDRAG
BOOL APIENTRY DrgVerifyRMF( PDRAGITEM pditem,
                             PSZ pszMech,
                             PSZ pszFmt) ;
```

<i>Parameter</i>	<i>Description</i>
pditem	Pointer to a DRAGITEM structure
pszMech	Text string containing the rendering mechanism
pszFmt	Text string containing the rendering format
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

In the two text strings, there will be a mechanism/format pair that will be compared to the contents of the text string indicated by the handle *hstrRMF* of the dragged object. The latter can present several mechanism/format pairs, all to be examined by *DrgVerifyRMF()*. With *DrgVerifyNativeRMF()*, instead, you consider only the first mechanism/format pair, which is generally known as the native one.

```
#define INCL_WINSTDDRAG
BOOL APIENTRY DrgVerifyNativeRMF( PDRAGITEM pditem, PSZ pszRMF) ;
```

<i>Parameter</i>	<i>Description</i>
pditem	Pointer to a DRAGITEM structure
pszRMF	String containing a format/mechanism pair for rendering
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

An application that needs to interact with the user through a drag & drop mechanism will have to intercept the *DM\_DRAGOVER* message. The information contained in *DRAGINFO* and in the associated *DRAGITEM* structures can originate directly from a *WPS* object, or from some other third party program. Whatever the source, the receiver must first decide the nature of the objects associated with the mouse pointer. Then the code needs to examine in detail the kind, format, and mechanism of rendering. Only when all these tests have been passed successfully will it be possible to concentrate on the operation being performed: copying, shadowing, moving, creating, or even something else (*DO\_UNKNOWN*). It is during this phase of processing that the application will have to prepare the information for the return value of the *DM\_DRAGOVER* message.

## *Changing the Cursor's Look*

When preparing data before calling *DrgDrag()* you will have to examine the *DRAGIMAGE* structure to establish what image will be displayed during dragging. A window that received the *DM\_DRAGOVER* is allowed to entirely change the image associated with the cursor, and it will be able to do so by filling in the members of a *DRAGIMAGE* structure, then calling the *DrgSetDragImage()* function to activate the new information.

```
#define INCL_WINSTDDRAG
BOOL APIENTRY DrgSetDragImage( PDRAGINFO pdingfo,
                                PDRAGIMAGE pdimg,
                                ULONG cding,
                                PVOID pRsvd) ;
```

<i>Parameter</i>	<i>Description</i>
pdinfo	Pointer to a DRAGINFO structure
pdimg	Pointer to a DRAGIMAGE structure
cdimg	Number of DRAGIMAGE structures
pRsvd	Reserved

<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

When the mouse's cursor moves over a new window, the returned image will always be the one originally planned for when calling *DrgDrag()*, except for the possible changes that can be forced by calling *DrgSetDragImage()*. In general, it will not be necessary to change the appearance of the images associated with the mouse's cursor, at least when the application's dragging operation is performed with respect to WPS or other programs. However, dragging can be very convenient and productive even within your own application, usually for the sake of simplifying operations. In this case, you are free to interpret the standard rules in whatever way you deem appropriate.

To change the mouse's pointer (not the associated image) you will have to resort to the *DrgSetDragPointer()* function. Almost invariably, the images used in this case are the standard ones listed in Table 4.5 of Chapter 4.

```
#define INCL_WINSTDDRAG
BOOL APIENTRY DrgSetDragPointer( PDRAGINFO pdingfo, HPOINTER hpPtr) ;
```

<i>Parameter</i>	<i>Description</i>
pdinfo	Pointer to a DRAGINFO structure
hpPtr	Handle to the mouse's pointer

<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

*DrgSetDragPointer()* will rely upon the services of *WinQuerySysPointer()*, as in the following code fragment concerning a window that will not be able to receive dragged objects:

```
...
DrgSetDragPointer( pdrginfo, WinQuerySysPointer( HWND_DESKTOP,
                                                SPTR_ILLEGAL, FALSE)) ;
...
```

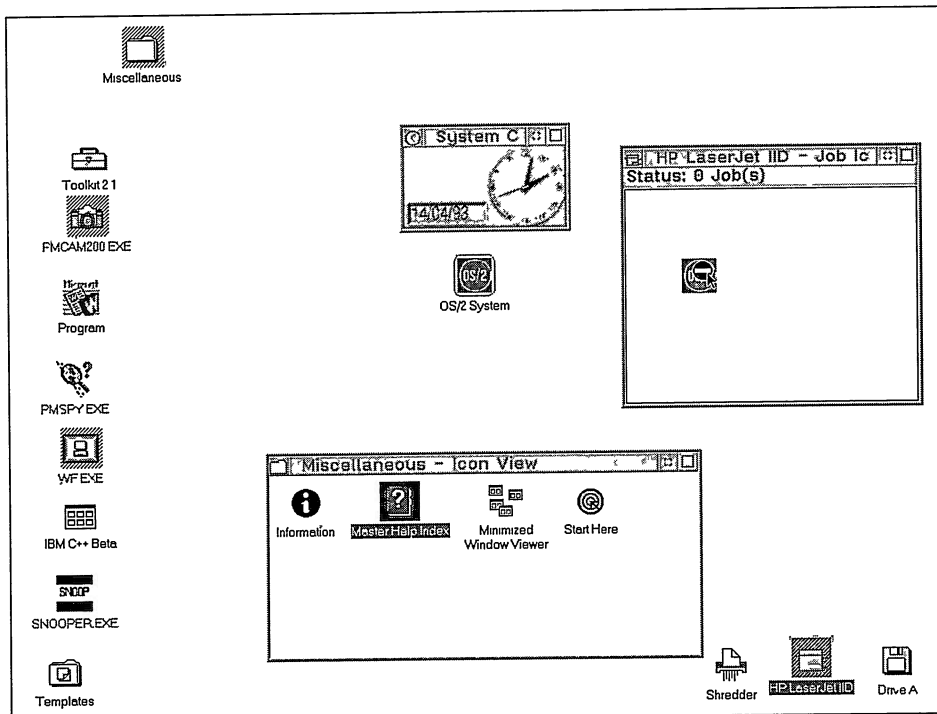
Resorting to *DrgSetDragPointer()* in the code dealing with the *DM\_DRAGOVER* message is necessary if you want to display a nonstandard cursor. *DrgDrag()* will take care of displaying a low intensity arrow, or the arrow/interdiction pair sign according to

the values returned by the processing of the DM\_DRAGOVER message. In writing the code supporting the acceptance of objects originating from WPS through dragging, you must use the standard internal icons of PM.

## Acceptance Feedback

One of the tasks when processing the DM\_DRAGOVER message is making it clear to the user where the dragged object may be dropped at any given moment. At the present stage, in WPS, a very simple rule is followed. Apart from the actual acceptance or not of the contents of the dragged object, the window underlying the mouse pointer is emphasized by a thin outline border. It will only be the shape of the mouse's cursor that will give any clues at all regarding the possible interactions (Figure 12.5).

An alternate solution, preferable in some situations, is to display the outline border *only* if there can be a total acceptance of the contents of the dragged object. Whatever solution is implemented (i.e., following the behavior of WPS when implementing a drag & drop that involves interactions outside the originating application), the operations that must be performed are few and simple. It will be necessary, first of all, to obtain a handle to the presentation space through *DrqGetPS()*.



**Figure 12.5** In WPS the receiving object in a drag & drop operation will always have an emphasizing outline border, even if there are incompatible types involved in the operation.

```
#define INCL_WINSTDDRAG
HPS APIENTRY DrgGetPS( HWND hwnd) ;
```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window underlying the mouse's hot spot
<i>Return Value</i>	<i>Description</i>
HPS	Handle to the presentation space or NULLHANDLE in case of error

The syntax of *DrgGetPS()* is the same as that of *WinGetPS()*, which was examined in Chapter 3. Similarly, the same considerations made for *WinReleasePS()* apply to *DrgReleasePS()*. Displaying an emphasizing outline border is a simple matter, once you have a handle to a presentation space.

```
#define INCL_WINSTDDRAG
BOOL APIENTRY DrgReleasePS( HPS hps) ;
```

<i>Parameter</i>	<i>Description</i>
hps	Handle to a presentation space
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The objects of WPS will present a thin outline border near the border of the client area of a folder, or around an icon (object). The appearance of a connecting line between the mouse pointer and the starting object when creating a shadow, is an operation assigned to *DrgDrag()*, and does not interact in any way with the activity painting performed in the target window.

The emphasizing of the target will take place corresponding not only to the *DM\_DRAGOVER* message; even *DM\_DROP* and *DM\_DRAGLEAVE* are concerned in this operation. The message *DM\_DROP* will be issued when the user terminates the dragging operation. After retrieving all the information from *DRAGINFO* and from the associated *DRAGITEM* structures, you will also have to remove the emphasizing border, which will still be present in this situation, whatever approach was followed in making it appear. The *DM\_DRAGLEAVE* message reaches the window as soon as the mouse has exited it. It is always the *DrgDrag()* function that issues this message when it detects that the handle of the window underlying the hot spot is different from the one obtained from the most recent *WM\_MOUSEMOVE*. Therefore, this is also the ideal situation for removing the emphasizing border and restoring the original look of the window.

## ***Return Value of DM\_DRAGOVER***

Once the test on the object type and on the rendering mechanism/format pair has been completed with respect to the kind of supported operation, processing of the *DM\_DRAGOVER* message can be terminated by building a return value of the type *MRESULT*, starting from a couple of *USHORT*. In the first one you will need to enter one of the defines listed in Table 12.10.

**Table 12.10 Values in the First USHORT to Be Sent in Response to the DM\_DRAGOVER Message**

<i>Code</i>	<i>Value</i>	<i>Description</i>
DOR_NODROP	0x0000	Rejects the object(s) being transferred during drag & drop operation even if its type and actions are supported, simply because the current position is not correct.
DOR_DROP	0x0001	Accepts the object(s) being transferred in the drag & drop operation.
DOR_NODROPOP	0x0002	The object(s) are supported, but not the involved operation.
DOR_NEVERDROP	0x0003	Rejects the object(s) for an indefinite time. The window has not received the DM_DRAGOVER message for subsequent mouse movements. It will receive them again when the mouse re-enters the window's space (after having exited earlier).

The second USHORT encodes the operation (D0\_ define) supported by the window for that specific object. The return value is constructed resorting to the macro MRFROM2SHORT.

```

...
case DM_DRAGOVER:
{
    ...
    return MRFROM2SHORT( DOR_DROP, D0_COPY ) ;
}
...

```

This code fragment has been presented here for the purpose of illustration. In fact, it is quite unlikely that MRESULT will ever be built starting with a pair of defines. Instead, the DOR\_ and D0\_ defines are two identifiers given appropriate values within the code dealing with DM\_DRAGOVER, considering the type of operation implied by the dragging and the kind of interaction supported by the program. The programmer can play a prominent role here by deciding how to interact with the object associated with the mouse.

Let's examine what happens when the user drags a *Data File* object produced by a data file template over an application that is capable of interacting with a drag & drop. In the member `usOperation` of `DRAGINFO` you will find the value `D0_DEFAULT`. WPS will then indicate that dragging a document will be handled like a default operation. The default operation will change according to the final receiver and other specific conditions. For instance, the default operation for a file that is dragged over an object

like a *Drive* means a *move* if it is within the same disk; the same operation means *copy* when two different drives are involved.

The DOR\_ define to be returned is thus a direct consequence of the level of interaction between an application and the dragged objects. The define DOR\_NEVERDROP can be called only if you are absolutely certain that the window being dragged over will never be able to accept anything. Issuing a DOR\_NEVERDROP will prevent the receipt of any subsequent DM\_DRAGOVER messages, even if the user presses the keys CTRL and/or SHIFT to modify the operation. If an operation is to support drag & drop, then it will issue DOR\_NEVERDROP only when the type, format, and mechanism of rendering is totally incompatible with its own capabilities.

DOR\_NODROPOP is the correct choice when incompatibility is limited to the operation indicated at that specific moment. Issuing DOR\_NODROPOP will not preclude issuing DM\_DRAGOVER after pressing some other keys, like CTRL and/or SHIFT. DOR\_NODROP can be used when all compatibility tests have been successfully met but the current mouse position does not identify an area that is capable of accepting the involved objects. The message DM\_DRAGOVER will continue to be issued after any movement of the mouse or any key presses.

Needless to say, DOR\_DROP must be used only when everything is just right!

## *Frame/Client and Dragging*

A PM window is actually a collection of several windows. This is no novelty, but this feature affects even drag & drop operations. When the mouse passes over the border of a window during a drag & drop operation, it sends the message DM\_DRAGOVER and then DM\_DRAGLEAVE to the *frame window*. The latter is not always subclassed, and thus the whole control of the message flow will reside in the window procedure of the class WC\_FRAME. The practical consequence is the passing of the two messages to the *client window*. It is unlikely that you will ever even think of implementing drag & drop operations on the *frame*. If the messages are not caught in the frame procedure, then the processing employed by the client procedure also becomes the one adopted by the frame. If the client supports that specific kind of dragging, the frame will adapt—at least from a visual point of view. WPS follows precisely this approach as far as folders are concerned. In practice the sequence of messages that reaches the client's window procedure when the mouse enters is like this:

```
DM_DRAGOVER    sent to the frame
DM_DRAGLEAVE   sent to the frame
DM_DRAGOVER    sent to the client
```

while when the mouse exits the message flow becomes:

```
DM_DRAGLEAVE   sent to the client
DM_DRAGOVER    sent to the frame
DM_DRAGLEAVE   sent to the frame
```

The main disadvantage of processing these DM\_ messages twice is the flickering of the emphasizing border, which is removed first by DM\_DRAGLEAVE, and then appears immediately after the first DM\_DRAGOVER addressed to the *client*. By preventing DM\_DRAGOVER and DM\_DRAGLEAVE from passing from the frame to the client you can avoid any duplicate efforts in the application. You will have to subclass the frame, and then make sure that the two messages don't ever reach the processing implemented in the window procedure of the WC\_FRAME class. Here's the code:

```
...
case DM_DRAGOVER:
case DM_DRAGLEAVE:
    return 0L ;
...
```

In this way, the frame will always respond negatively to a drag & drop operation. Naturally, it does not make sense to perform subclassing for these two lines of code alone. In WPS objects (like folders) the dragging of an object is intercepted twice, by the frame first and then the client.

## Receiving Objects

When the user releases the right mouse button over a window that has previously indicated it is capable of accepting an object, the message DM\_DROP will be issued.

DM_DROP	0x032F	<i>Description</i>
mp1	PDRAGINFO	Pointer to a DRAGINFO structure
mp2	ULONG	Reserved
Return Value	ULONG	Reserved

The code handling the DM\_DROP message will first of all transform the contents of mp1 into PDRAGINFO pointer.

```
case DM_DROP:
{
    PDRAGINFO pdrginfo = (PDRAGINFO)mp1;
    ...
}
```

The pointer to the DRAGINFO structure is the basis for the subsequent processing that will read the contents of the many associated DRAGITEM structures (in our example there is just one). However, transforming the contents of mp1 into a PDRAGINFO pointer is not the only operation that is needed. You will also need to obtain permission to access this memory area. In fact, dragging will often occur between different processes, all having a private address space of their own, which cannot by definition be shared. Fortunately, you are not confined to using the *Dos* memory management functions, and you can simply call *DrgAccessDragInfo()* :

```

...
PDRAGITEM pdrgitem ;
...
if( !DrgAccessDraginfo( pdrginfo))
{
    WinAlarm( HWND_DESKTOP, WA_NOTE) ;
    break ;
}
...

```

The count of associated DRAGITEM structures, and the access to the first one of these is the result of the calls to *DrgQueryDragitemCount()* and *DrgQueryDragitemPtr()*:

```

...
// # of DRAGITEM structures
ulItem = DrgQueryDragitemCount( pdrginfo) ;
pdrgitem = DrgQueryDragitemPtr( pdrginfo, 0L) ;
...

```

From this point on, the behavior caused by processing the DM\_DROP message is not standardized and changes according to the specific needs of the program. Nevertheless, the basis of the subsequent operations consists of accessing all information directly available in the DRAGITEM structure or the text strings indicated by the special handles.

The code block handling DM\_DROP is then terminated by a call to *DrgFreeDraginfo()* for decreasing the usage count of the memory area allocated at the beginning of the drag & drop operation.

```

...
// free the info block
DrgFreeDraginfo( pdrginfo) ;
...

```

## *Dragging the Titlebar Icon*

A thorough knowledge of the rules governing drag & drop is fundamental for implementing applications that integrate well with the world of WPS. In Chapter 13 we will put together all pieces of the puzzle and implement some real-world examples. For the moment, though, we still need to gain more experience with the single components of the API. For instance, let's examine the titlebar icon. In the WPS model it is an image that qualifies the kind of object with which the user is interacting, and is not just a standardized meaningless bitmap. Some applications that try to match the development model of WPS, use the titlebar icon for still other purposes. This is the case, for example, of EPM.EXE, the advance system editor of OS/2. If you keep the right mouse button pressed over the titlebar icon, and then move the mouse, you will see a typical drag & drop effect. The dragged icon is exactly the one appearing on the titlebar, and corresponds to a stylish document (Figure 12.6).

This behavior is defined as titlebar dragging, and is fundamental for implementing many kinds of interaction between the user and a WPS-compliant application. Releasing



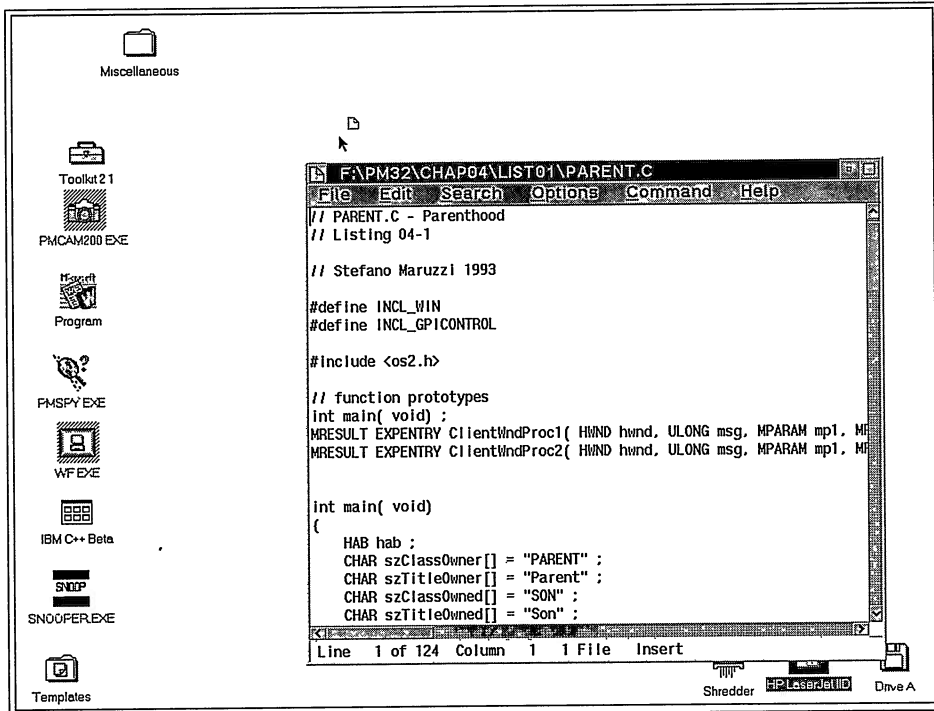


Figure 12.6 Dragging the titlebar icon in EPM.EXE.

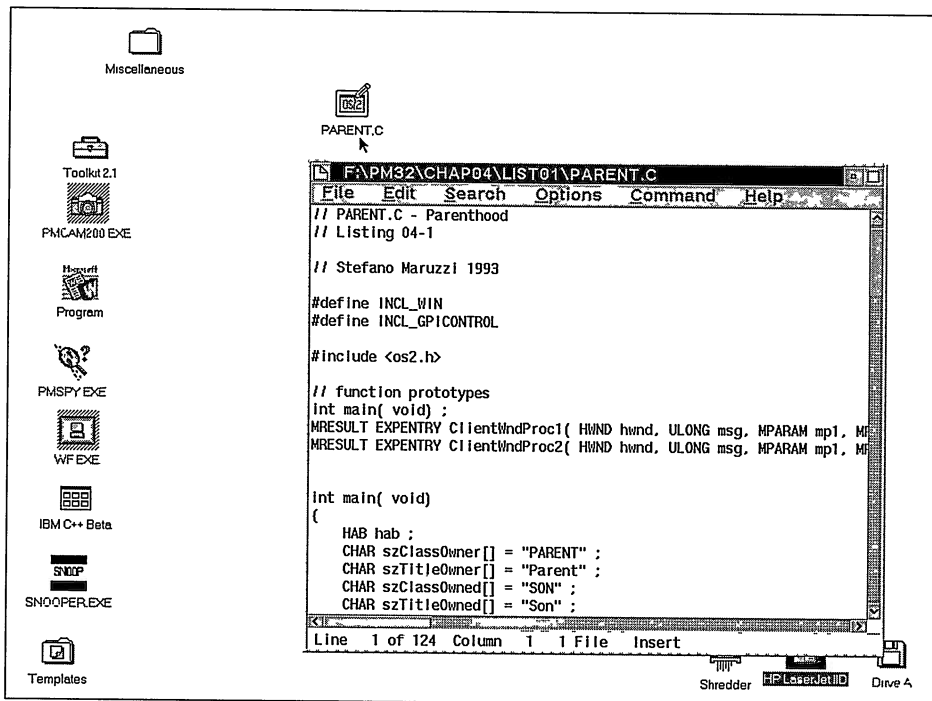
the right mouse button over the desktop or over any other container will cause a copy of the file being edited to appear there (Figure 12.7).

Let's see how we can build this kind of behavior in our applications. For this purpose we will build two sample applications, DRAG and DROP, that enclose respectively the logic for initiating *dragging* and for receiving *dropping*.

## DRAG: Preparing Data

First of all, let's think about what happens when the user presses the right mouse button over the titlebar icon. Usually, this will involve the appearance of the associated drop-down menu. So we will have to make sure this does not happen if the user keeps the button depressed. The operation will inevitably involve *subclassing* the window in order to be able to intercept in a function of your own the message flow generated by the user. For this action, the sequence of messages is the following one:

```
WM_BUTTON2DOWN
WM_BUTTON2UP
WM_BUTTON2CLICK
WM_CONTEXTMENU
```



**Figure 12.7** Dragging the titlebar icon and then releasing the mouse button over the desktop have generated a new object.

The display of the drop-down menu will happen immediately after this, with the sending of the message `MM_STARTMENU`. This will happen even in `DRAG`; but if the button is not released, and the mouse is moved just a fraction of an inch, you will find yourself dealing with a series of events that indeed qualify the action as dragging. In this case, the messages are:

```

WM_BUTTON2DOWN
WM_BUTTON2MOTIONSTART
WM_BEGINDRAG
WM_BUTTON2MOTIONEND
WM_ENDDRAG

```

The only common message is the first one. The corresponding action is activating the application's titlebar. That event is simulated inside the code block handling `WM_BUTTON2DOWN`, and thereby preventing the message from being subject to default processing.

```

...
case WM_BUTTON2DOWN:
    if( WinQueryActiveWindow( HWND_DESKTOP) != PAPA( hwnd))
        WinSetActiveWindow( HWND_DESKTOP, PAPA( hwnd)) ;
    return (MPARAM)0L ;
...

```

The objective of this solution is activating the titlebar (the window) even when the user is about to drag the titlebar icon. This is essentially all that needs to be done to activate the dragging of the icon. We still need to define the various details that will prepare the actual dragging inside the code handling the `WM_BEGINDRAG` message.

The DRAG application provides a window of the class `WC_MLE`, overlapping the client window. As a result of dragging the titlebar we will allow the user to save in a file whatever text has been typed in. Listing 12.1 presents the code of DRAG, the sample listing specifically devised for illustrating this first part of drag & drop techniques and the interaction with WPS.

Initially, focus is on the *mle* precisely for inviting the user to type in some text. Dragging the titlebar icon when the *mle* is empty will not produce any effect. This behavior corresponds to the rationale of not allowing the user to generate any new files unless there is some text to save.

In order to check the contents of the *mle*, we resort to a specific function called *SaveText()*, which will also generate a temporary file in the current directory. The *SaveText()* functions will accept as their parameter the *mle*'s handle and the name to give to the file. The computation of the size of the text present in the *mle* is required through the `MLM_QUERYTEXTLENGTH` message, which will return the number of characters present in the window. This value, though, will have to be increased by the number of new lines that are present. In fact, the line termination sequence involves the pair of characters `\n\r`. The message `MLM_QUERYTEXTLENGTH` does not take this aspect into account. The text's size is thus corrected with the value produced by sending the `MLM_QUERYLINECOUNT` message:

```
...
// get document size
ulLines = (ULONG)WinSendMessage( hwndMLE, MLM_QUERYLINECOUNT,
                                0L, 0L );

if( !ulLines)
    break ;

ulBufferSize = (ULONG)WinSendMessage( hwndMLE, MLM_QUERYTEXTLENGTH,
                                       0L, 0L );

ulBufferSize += ulLines - 1 ;
ulSize = ulBufferSize ;
...
```

*SaveText()* will return immediately and signal an error if the *mle* is empty. Any attempt to drag will thereby be prevented. If this test is met, we can proceed by allocating one or more memory pages, and transferring the contents of the *mle* there during the export phase, which is controlled by the message `MLM_EXPORT`.

```
...
// allocating memory
if( DosAllocMem( (PPVOID)&pBuffer, ulBufferSize,
                PAG_WRITE | PAG_COMMIT))
{
    WinAlarm( HWND_DESKTOP, WA_ERROR) ;
}
```

```

    return FALSE ;
}

// define export buffer
WinSendMessage( hwndMLE, MLM_SETIMPORTEXP,
                MPFROMP( pBuffer), MPFROMLONG( u1Size)) ;
// export text
u1Export = (ULONG)WinSendMessage( hwndMLE, MLM_EXPORT,
                                   MPFROMP( (PIPT)&10offset),
                                   MPFROMP( &u1Size)) ;

...

```

When using the message `MLM_EXPORT` you must be careful with the two parameters `mp1` and `mp2`. In `mp1` there must appear the address of an identifier initialized to 0L for indicating the starting position of the copy of the contents of the *mle*. In the second `MPARAM`, instead, you must indicate the size of the buffer to copy. Here comes the trap! Sending the `MLM_EXPORT` message implies that `mp2` will contain the number of characters to be exported decreased by the number of characters actually exported. The net result is the loss of `u1Size` from the size of the text transferred from the *mle* to the auxiliary buffer. This information will subsequently be used in writing the data to a file with `DosWrite()`, and thus explains the presence of `u1BufferSize` containing the overall file dimension, unchanged from the initial assignment.

Drag & drop operations are essentially based on files. In the case of DRAG, the information present in the *mle* is not stored on a disk in the system. Before the actual dragging starts, you will have to transfer to the disk the contents of the memory buffer in order to generate a temporary file. When writing the code for opening the temporary file that will be the foundation for the titlebar dragging operation, you must be extremely careful about the flags you use:

```

...
if( rc = DosOpen( pszFullFile, &hfile,
                 &u1Action,
                 u1BufferSize,
                 FILE_NORMAL,
                 OPEN_ACTION_CREATE_IF_NEW |
                 OPEN_ACTION_REPLACE_IF_EXISTS,
                 OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYWRITE |
                 OPEN_FLAGS_SEQUENTIAL,
                 NULL))

...

```

In fact, although the preliminary operations of this kind of drag & drop initially involve the DRAG application, it is almost certain that dropping will involve some other WPS object, or even the desktop window. The presence of the attributes `OPEN_ACTION_CREATE_IF_NEW` and `OPEN_ACTION_REPLACE_IF_EXISTS` will allow for the creation of a new file if it does not already exist in the target object, or will present a dialog asking the user what should be done if there already exists an object with the same name.

The *SaveText()* function completes its work by writing to the temporary file the text retrieved from the *mle*, and then closing the file by calling *DosClose()* (and thereby also destroying the allocated memory area). Only at this point can the true dragging operation start, by filling in appropriate information in the DRAGINFO, DRAGITEM, and DRAGIMAGE structures. The default operation will be the default (DO\_DEFAULT). The drag & drop supported by OS/2 always refers to a file, references to files (*shadow*), or a pointer to a file (the objects in the *System Setup* folder, for instance). That's why in the titlebar dragging example you are first forced to create a temporary file that you will then delete once the operation is finished. The final outcome is spectacular, as you can see in Figure 12.8.

In Figure 12.9 you can see how the document created on the desktop is the same as the text typed into the *mle* of the DRAG application.

Summarizing, the operations provided for by the DRAG program are the following:

- Subclassing the titlebar icon
- Transferring the text present in the *mle* into a temporary file in the current directory
- Allocating and setting up the DRAGINFO, DRAGITEM, and DRAGIMAGE structures
- Calling the *DrgDrag()* function
- Destroying the temporary file

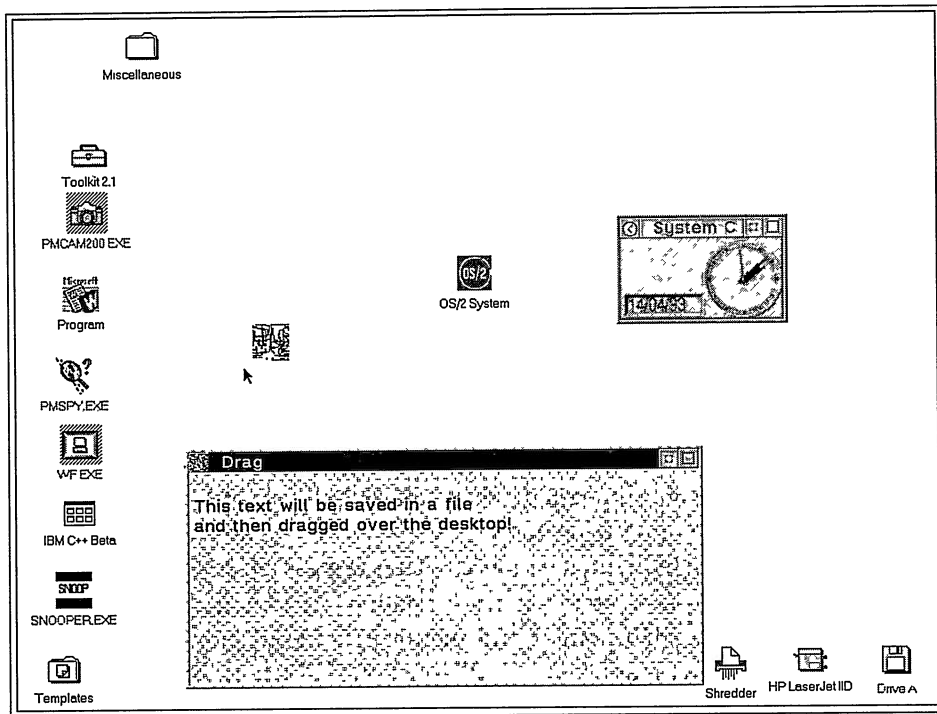


Figure 12.8 The DRAG application while titlebar icon dragging is being executed.

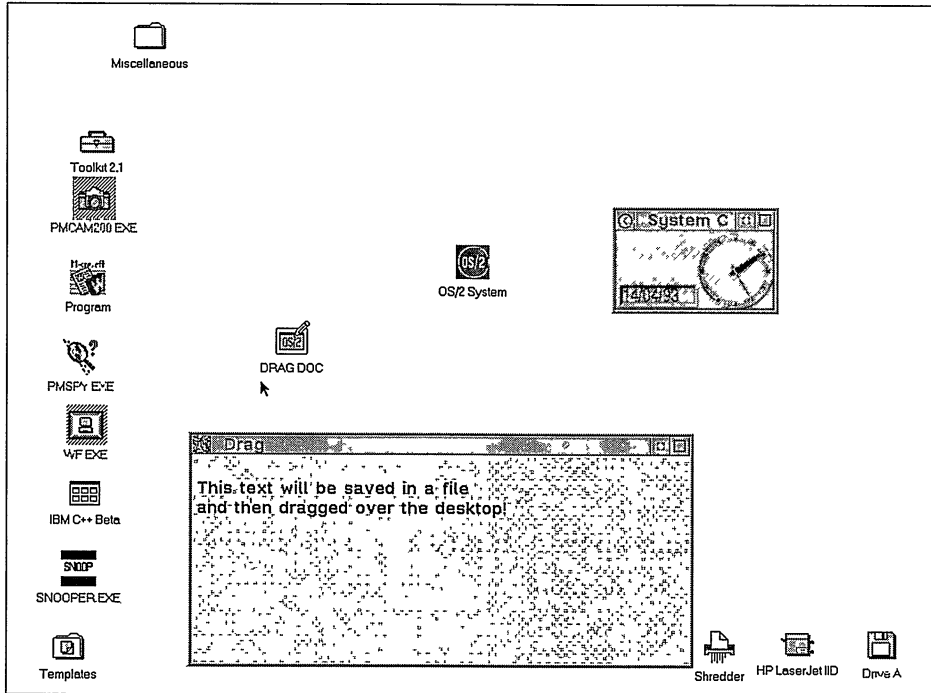


Figure 12.9 Final result of dragging the titlebar icon.

Let's now examine the other half of the drag & drop by implementing the DROP application, which is capable of accepting text objects originating from any other kind of source (application or container) present in the system.

## DROP: Accepting an Object

The most interesting aspect of developing an application capable of receiving information delivered by a drag & drop operation is no doubt its appearance. In fact, when the user drags an object over a container, there will often be variations in the mouse's cursor shape (like an interdiction sign appearing where the object cannot be accepted) or of the underlying window (emphasizing border). All these changes are the outcome of processing information associated with the mouse when the message `DM_DRAGOVER` is received.

The first operation is accessing the information describing the dragging, first by dereferencing a `PDRAGINFO` pointer and then a `PDRAGITEM` pointer. By means of the `PDRAGITEM` pointer and the appropriate API function of PM, the receiving program is able to verify the type, format, and rendering mechanism that was specified by the sender during the preparatory phases of dragging. In the specific case of the DROP application, the verification of this information is based on the following code fragment:

```

...
// verify the type
if( !DrgVerifyType( pdrgitem, DRT_TEXT))
{
    WinAlarm( HWND_DESKTOP, WA_ERROR) ;
    break ;
}
// verify the mechanism and the format
if( !DrgVerifyRMF( pdrgitem, "DRM_OS2FILE", DRF_TEXT))
{
    WinAlarm( HWND_DESKTOP, WA_ERROR) ;
    break ;
}
...

```

The test will search for a type and a format with *DrgVerifyType()* and *DrgVerifyRMF()* rather than with the specific *DrgVerifyNativeType()* and *DrgVerifyNativeRMF()* functions, because it is never possible to be certain about the source of dragging. This information must involve the type `DRT_TEXT`, while the supported rendering mechanism and format are `DRM_OS2FILE` and `DRF_TEXT`. These are typical characteristics of a generic file in ASCII format.

Once consistency of the information associated with the mouse has been verified, the DROP application will evaluate the kind of operation proposed by the sender for setting up the most appropriate visual feedback, and then make sure it will be able to support the proposed operation:

```

...
switch( pdrginfo -> usOperation)
{
    case DO_DEFAULT:
        ...
        break ;

    case DO_UNKNOWN:
        ...
        break ;

    case DO_COPY:
        usDOR = DOR_DROP ;
        usDO = DO_COPY ;
        break ;

    case DO_MOVE:
        usDOR = DOR_DROP ;
        usDO = DO_MOVE ;
        break ;

    case DO_LINK:

```

```

    ...
    break ;

case DO_CREATE:
    ...
    break ;
}
...

```

In the identifiers `usDOR` and `usD0`, there will appear, respectively, the defines `DOR_` and `D0_` appropriate for the proposed operation and application-specific features. The assignment of the define `DOR_DROP` to the `usDOR` identifier will also determine the type of emphasis implemented by the program:

```

...
// get the presentation space handle
hps = DrgGetPS( hwnd) ;

if( usDOR == DOR_DROP)
{
    WinDrawBorder( hps, &rc, 1L, 1L,
                  CLR_BLACK, CLR_WHITE, DB_STANDARD) ;
}
else
{
    GpiErase( hps) ;
}
// release the handle
DrgReleasePS( hps) ;
...

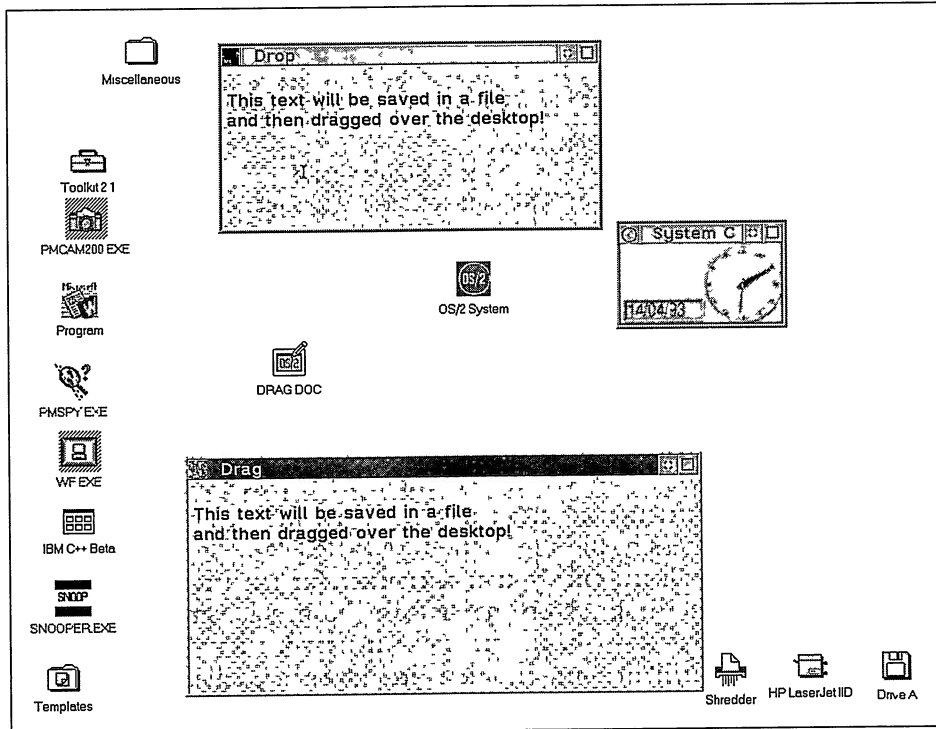
```

The logic followed by `DROP` is rather limited. A thin border will appear inside the client window if the proposed operation is supported. If it is not, the contents of the client window will be erased with `GpiErase()`. All these operations take place directly on the client, and thus enable the reception and handling of the `DM_` message generated by the drag & drop operations. When the user releases the right mouse button, the `DM_DROP` message will be received. In this case, after appropriately accessing the information about the dragging, the client will be overlapped by the `WC_MLE` class window that was previously created when the `WM_CREATE` was received, but never displayed (Figure 12.10).

## Intercepting `DM_DROP`

In the code handling `DM_DROP` the consistency tests of the information associated with the mouse are not repeated for the receiving application (which would be redundant). Rather, processing this message will focus on retrieving the text strings regarding the





**Figure 12.10** Effect of dragging the contents of the DRAG application into the DROP application.

source container (the directory where the starting document is found), the name of the source file, and the name of the target file in the destination container. In the case of DROP, the application is not a container, but a program. Although the drag & drop is used mainly at level of WPS objects (between containers), nothing prevents you from resorting to this protocol internally in the same application or, as in our example, between two different EXEs. In the case of the DRAG and DROP sample applications, both are executable without any kind of container object. The generation of the information to be dragged and its processing is handled completely in the window structures of the two programs without any problems.

This reference to the WC\_CONTAINER class is not out of place, as you will see later, because container windows are specially designed for supporting in a simple and direct way drag & drop operations. It is helpful to remember that even the *desktop* is a window belonging to the WC\_CONTAINER class. The lack of a container will not pose any design problems for the DRAG application, or for the DROP application, except for the impossibility of displaying the object being dragged as an icon (the text is automatically exposed in the *mle*). Processing the DM\_ messages is appropriately handled on the basis of the windows involved in the two applications.

```

...
// retrieve the source container name
DrgQueryStrName(    pdrgitem -> hstrContainerName,
                   sizeof( szContainer), szContainer) ;

// retrieve the source file name
DrgQueryStrName(    pdrgitem -> hstrSourceName,
                   sizeof( szSourceFile), szSourceFile) ;

// retrieve the file name
DrgQueryStrName(    pdrgitem -> hstrTargetName,
                   sizeof( szFileName), szFileName) ;

// create full pathname
strcat( szContainer, szSourceFile) ;
...

```

The complete pathname constructed in the `szContainer` identifier allows you to access the file created by DRAG when the whole process was started. Once the file is open, DROP will simply transfer its contents into the *mle*. The technique used in the DROP example for accessing the information associated with the mouse is not the only one possible. As we will see, once you have checked for consistency between the two objects concerned in the drag & drop, the choice of the mechanism used by the receiver for accessing the data is up to the programmer. In practice, any IPC tool provided by OS/2 can be employed, like communication through a shared file. Not always, though, will drag & drop involve two portions of code written by the same programmer. In fact, it is rather unlikely that the user will ever be interested in dragging any kind of object toward any other kind of object without taking into account their nature and origin. Therefore, it is always a good practice to be sure that the supported actions work well even with external objects. For example, the DROP application, as you can see in Figure 12.11, is able to show inside the *mle* the text contained in a document accommodated in the desktop, and originally produced from a *Data File template*.

The transfer of contents from DRAG to DROP can follow two paths:

- Dragging the titlebar icons from DRAG to DROP
- Creating a document in a WPS container (even on the desktop), and subsequent dragging of this document into DROP

The implementation of titlebar dragging brings with it some very interesting implications. In practice, the operation of dragging the icon to the desktop is equivalent to saving a document, which can easily be extended into a *Save As* operation. In Chapter 13 we will further explore the potential of this form of dragging.

Listing 12.2 contains the source code of the DROP application.




---

## Drag & Drop and Valuesets

The class `WC_VALUESET` is an ideal tool for building useful application components. For instance, you might implement a window like a panel containing several native

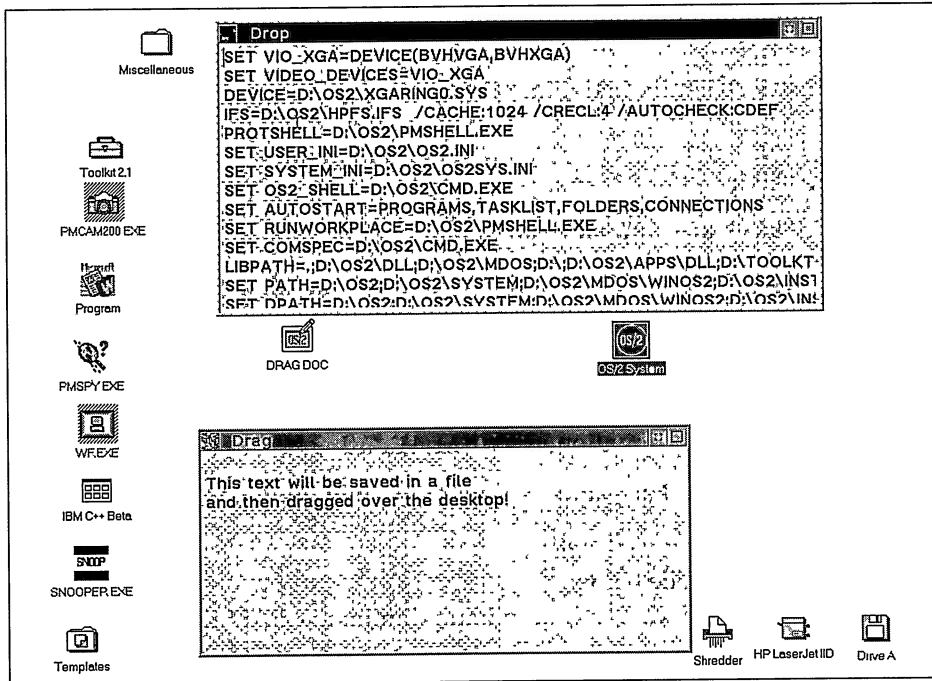


Figure 12.11 The DROP application is capable of displaying data from a generic file present in the system.



icons or inserted there by means of a drag & drop operation. This kind of control panel allows you to embody in one sole object numerous functionalities of the application. That's what we will implement in the PANEL sample presented in Listing 12.3.

The styles of the class WC\_VALUESET do not allow you to define any kind of support for drag & drop actions. The attributes VIA\_DROPONABLE and VIA\_DRAGGABLE to represent, respectively, the possibility of accepting and originating drag & drop activities. When filling in each single cell in the panel, you will also assign the VIA\_DROPONABLE attribute by means of the VM\_SETITEMATTR message.

The *valueset* window will notify its owner about the drag & drop operations that affect it by sending appropriate notification codes. In the case of VN\_DRAGOVER, mp2 will contain a pointer to a VSDRAGINFO structure.

VN_DRAGOVER	123	<i>Description</i>
mp2	VSDRAGINFOpvsdraginfo	Pointer to a VSDRAGINFO structure.
Return Value	Reserved	

The identification of the affected cell is easy, because it is flagged by the pair of USHORT members. The VSDRAGINFO structure is then completed by a pointer to a DRAGINFO, as you might expect. The same behavior also pertains to the code VN\_DROP.

```

typedef struct _VSDRAGINFO
{ // vsdinfo
    PDRAGINFO pDragInfo ;
    USHORT usRow ;
    USHORT usColumn ;
} VSDRAGINFO ;

typedef VSDRAGINFO *PVSDRAGINFO ;

```

The PANEL application underlines a lack in the *Drg* API. Right now there is no way to retrieve the image (icon, bitmap, or polygon) associated with the mouse pointer. Usually drag & drop operations pertain to containers. WPS is a container, like all the folders in the system. I hope this bug will be fixed in future releases providing something like a *DrgQueryDraggedImaged()* or something similar.

I tried to bypass this limitation in the PM API using *WinLoadFileIcon()*:

```
HPOINTER WINAPI WinLoadFileIcon( PSZ pszFile, BOOL fIcon) ;
```

<i>Parameter</i>	<i>Description</i>
pszFile	File name
fIcon	TRUE to get a private icon image, FALSE a shared icon image

<i>Return Value</i>	<i>Description</i>
HPOINTER	Icon handle or NULLHANDLE in case of failure

*WinLoadFileIcon()* loads an icon from different sources. It first looks for the .ICON extended attribute, then it accesses the .ICO file if it is present in the current directory. Other sources of information are the different executables (OS/2 PM, Windows, OS/2 full screen, OS/2 windowed, and so on). If you don't plan to change the image, it is advisable to use FALSE; operations are faster and you don't need to recall *WinFreeFileIcon()*. Figure 12.12 shows the PANEL application after the user has dropped some objects in some cells.

---

## Drag & Drop and Listboxes

The PANEL application has some additional features as you can guess from Figure 12.12. The window to the right of the valueset is a owner-draw listbox. As described in Chapter 7, owner-drawn windows can contain any kind of object: text, bitmaps (icons), or a mix of the two ingredients. This specific listbox shows icons as its items. In the PANEL example there is no way to fill the listbox from inside the application. The only means is through drag & drop operations. So, let's follow the same approach used for the valueset. Select any object in WPS and drag it over the listbox (Figure 12.13).

The listbox behavior lets the user insert objects in any position. All you have to do is to release the right mouse button after reaching the appropriate location (Figure 12.14).

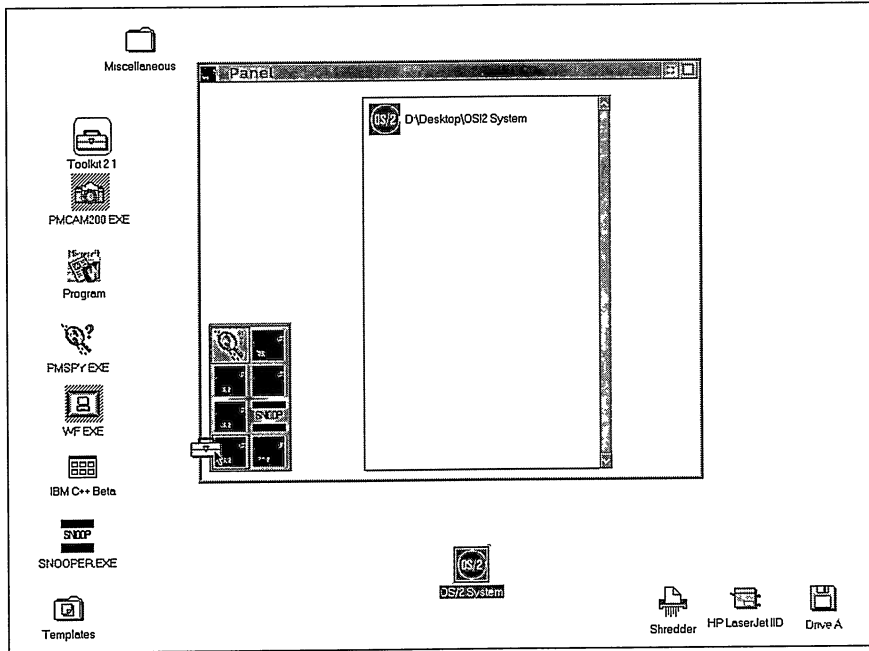


Figure 12.12 The PANEL application while it is engaged in a drag & drop operation.

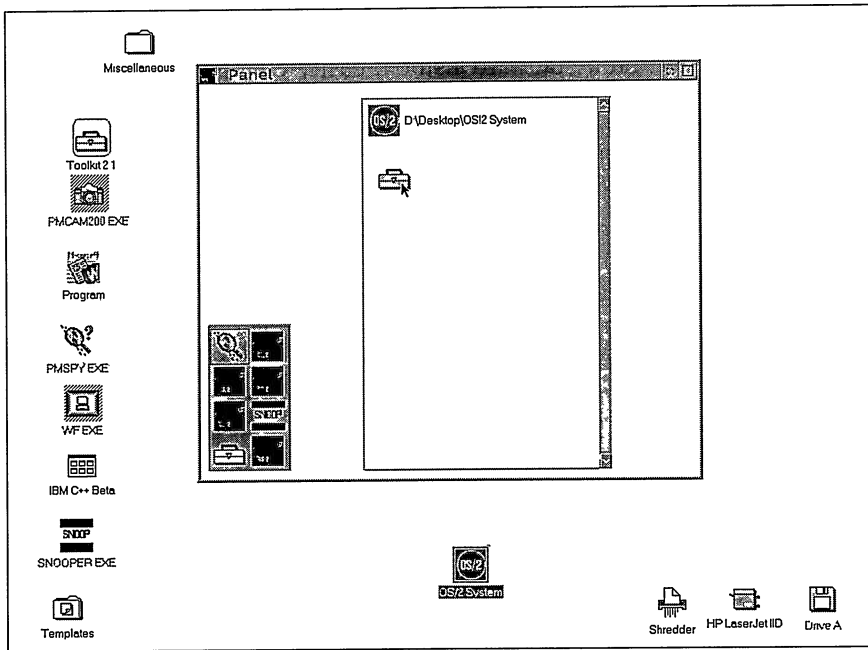


Figure 12.13 The insertion of an item is done through a drag & drop operation on the listbox.

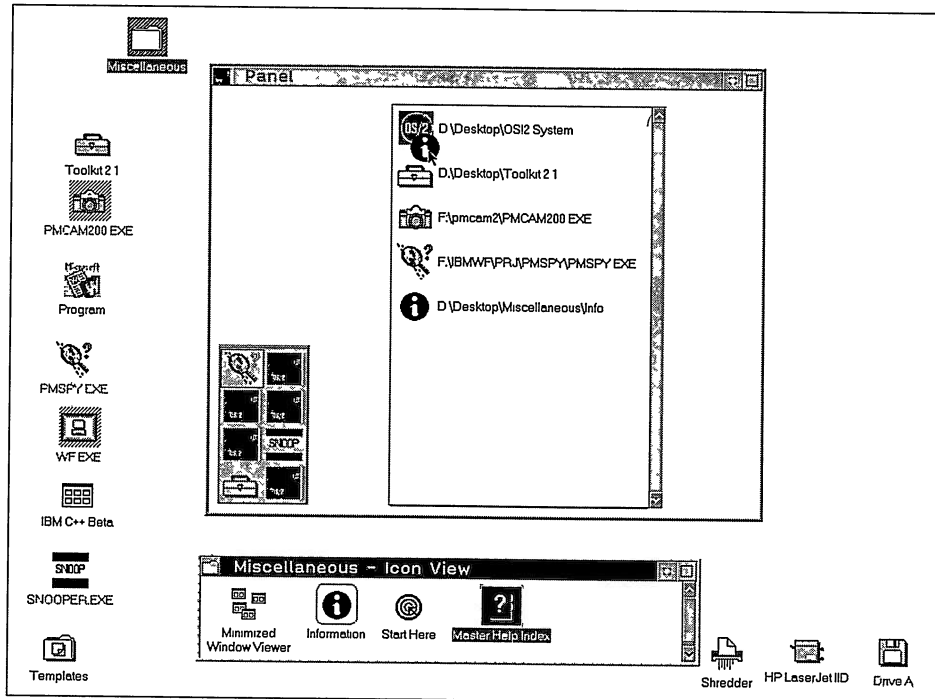


Figure 12.14 Objects can be placed everywhere in the listbox.

The example is based on a standard listbox, easily changeable into an extended selection or multiple selection. To achieve drag & drop interoperability with a listbox window the developer has to perform some preliminary actions. First of all, it must be subclassed. This is the only way to intercept DM\_ messages. The WC\_LISTBOX class is lacking any notification code to signal drag & drop events.

Basically, all the drag & drop code is identical to the one used for the valueset. The icon to display comes from the *WinLoadFileIcon()* function. The returned value is checked to trap a NULLHANDLE. In this case no icon is displayed and the application emits a beep. What is nice in this piece of code is the logic followed to determine the correct position for the incoming object, which is based on the following lines of code:

```
...
// deselect everything
WinSendMessage( hwnd, LM_SELECTITEM,
                MPFROMSHORT( LIT_NONE), MPFROMSHORT( FALSE) );

// how many items in the listbox?
sItems = (SHORT)WinSendMessage( hwnd, LM_QUERYITEMCOUNT, 0L, 0L );
// query the pointer position
WinQueryPointerPos( HWND_DESKTOP, &pt );
// transform the point in listbox coordinates
WinMapWindowPoints( HWND_DESKTOP, hwnd, &pt, 1L );
// simulate a left button down
```

```

WinSendMessage( hwnd, WM_BUTTON1DOWN,
                MPFROM2SHORT( (SHORT)pt.x, (SHORT)pt.y), 0L );
// simulate a left button down
WinSendMessage( hwnd, WM_BUTTON1UP,
                MPFROM2SHORT( (SHORT)pt.x, (SHORT)pt.y), 0L );
// query the selected item
sSel = (SHORT)WinSendMessage( hwnd, LM_QUERYSELECTION,
                              MPFROMSHORT( LIT_FIRST), 0L );

// if no selection the listbox might be either empty or
// the user is beyond the last item
if( sSel == -1)
    if( !sItems)
        // insert as the first one
        sSel = 0 ;
    else
        // the last one
        sSel = sItems ;

// deselect everything
WinSendMessage( hwnd, LM_SELECTITEM,
                MPFROMSHORT( LIT_NONE), MPFROMSHORT( FALSE));
...

```

First, we deselect all items in the listbox and then count the number of entries (`sItems`). This information is useful to determine the right position in the listbox. To find the mouse position on the listbox in terms of items (first, second, after the last, and so on) the application gets the mouse position (`WinQueryPointerPos()`), converts it into the listbox's coordinates (`WinMapWindowPoints()`) and then simulates the pressure of button 1. Now it is time to ask the listbox for the currently selected item. The final portion of this simple algorithm sets the `sSel` identifier to the appropriate value. When its value is `-1` it might depict two different situations. Either the listbox is empty or the user has dropped the object below the last item. To determine which is the case, evaluate `sItems`. If greater than zero, it means the new object has to be added at the very end of the list.

After accessing the drag & drop information the application passes the icon handle to the listbox's owner through the `WM_PASSICON` message and then simulates the insertion of a new item in the selected location.

```

...
hptr = WinLoadFileIcon( szContainer, FALSE );
// skip if no icon is available
if( !hptr)
    break ;

// pass the icon handle
WinSendMessage( PAPA( hwnd), WM_PASSICON, MPFROMLONG( hptr), 0L );

// insert new item
WinSendMessage( hwnd, LM_INSERTITEM,
                MPFROMSHORT( sSel), MPFROMP( "CIAO"));
...

```

The owner-drawn portion of the application is straightforward. All we have to do is to show the icon after erasing the background.

---

## Drag & Drop and Folders

Let's now make some changes to Listing 7.11 by adding support for drag & drop operations. The task isn't overly complicated because the WC\_CONTAINER class will itself provide a number of useful tools. Apart from the actual implementation of a container (in Listing 7.11 and in Listing 12.4 it is the client of an application), the first operation to be performed consists of taking advantage of the notification codes (Table 7.40). You will be concerned with retrieving the information conveyed by the CN\_INITDRAG, CN\_DRAGOVER, CN\_DRAGLEAVE, and CN\_DROP codes. Depressing the right mouse button combined with a movement of the mouse over any object present in the container will be perceived by the window as the start of a dragging operation. This is the moment when the owner will receive a CN\_INITDRAG. (This notification code, like all others, is notified through the WM\_CONTROL message. In the following description we will avoid this information in order to make the descriptive table more compact.)



CN_INITDRAG	107	<i>Description</i>
mp2	PCNRDRAGINIT pcnrdraginit	Pointer to a CNRDRAGINIT structure
Return Value	Reserved	

The CNRDRAGINIT structure is initialized automatically with information regarding the object on which the user is acting.

```
typedef struct _CNRDRAGINIT
{ // cdrginit
    HWND hwndCnr ;
    RECORDCORE pRecord ;
    LONG x ;
    LONG y ;
    LONG cx ;
    LONG cy ;
} CNRDRAGINIT ;

typedef CNRDRAGINIT *PCNRDRAGINIT ;
```

The owner will know the container's handle, and about all the information associated with a RECORDCORE structure (among which it will find the icon representing the object during dragging). The next pair of LONG identify the mouse's position on the screen, expressed in screen coordinates. The last two members give the displacement of the mouse's hot spot from the object. Almost invariably the notification codes of the WC\_CONTAINER class associated with drag & drop operations give the programmer full access to a RECORDCORE structure. This is great, but be careful! In the case of



CN\_INITDRAG, this will be the set of information that qualifies the record that was selected for dragging. For all other notification codes, it will instead be the set of data regarding the record over which the object is being dragged! This structuring of the API makes it a good idea to save the information regarding the RECORDCORE member of CN\_INITDRAG in a static storage class pointer. This will make it simpler to write the code dealing with all other notification codes.

```
...
static PAPPREC pappsource ;

case CN_INITDRAG:
{
    ...
}
...
```

Naturally, this is only the beginning of the work awaiting the programmer. Intercepting the notification codes alone won't be enough: You must also handle the dragging according to the normal rules described at the beginning of this chapter.

```
case CN_INITDRAG:
{
    PCNRDRAGINIT pcnrdraginit = (PCNRDRAGINIT)mp2 ;
    PDRAGINFO pdrginfo ;
    DRAGITEM drgitem ;
    DRAGIMAGE drgimage ;
    HWND hwndTarget ;
    PAPPREC papprec ;

    // allocate a DRAGINFO structure
    pdrginfo = DrgAllocDraginfo( 1L) ;

    pdrginfo -> hwndSource = pcnrdraginit -> hwndCnr ;
    pdrginfo -> usOperation = DO_COPY ;

    // initialize the DRAGITEM structure
    drgitem.hwndItem = pcnrdraginit -> hwndCnr ;
    drgitem.ulItemID = 100L ;
    drgitem.hstrType = DrgAddStrHandle( DRT_TEXT) ;
    drgitem.hstrRMF = DrgAddStrHandle( "<DRM_OS2FILE,DRF_TEXT>") ;
    ...
    drgitem.fsSupportedOps = DO_COPYABLE | DO_MOVEABLE ;

    if( !DrgSetDragitem( pdrginfo, &drgitem, sizeof( DRAGITEM), 0L))
        WinAlarm( HWND_DESKTOP, WA_ERROR) ;

    papprec = (PAPPREC)pcnrdraginit -> pRecord ;
    pappsource = papprec ;
    // allocate DRAGIMAGE structure
```

```

    drgimage.cb = sizeof( DRAGIMAGE) ;
    drgimage.cpt1 = 0 ;
    drgimage.hImage = pcnrdraginit -> pRecord -> hptrIcon ;
    ...
    drgimage.cyOffset = 0 ;

    hwndTarget = DrgDrag(   pcnrdraginit -> hwndCnr,
                           pdrginfo, &drgimage,
                           1L, VK_ENDDRAG,
                           NULL) ;
}
    break ;

```

The notification code CN\_DRAGOVER is received after any movement of the dragged object in the container. This time, the information conveyed by mp2 corresponds to the address of CNRDRAGINFO structure:

CN_DRAGOVER	103	<i>Description</i>
mp2	PCNRDRAGINFOpcnrdraginfo	Pointer to a CNRDRAGINFO structure
Return Value	Reserved	

In CNRDRAGINFO a pointer to a RECORDCORE structure appears, containing data about the underlying object, and a second pointer to a DRAGINFO structure. So the designer has access to the information that is typical of a drag & drop operation, as well as to those pertaining to the management of objects in a container.

```

typedef struct _CNRDRAGINFO
{ // cdrinfo
    PDRAGINFO pDragInfo ;
    RECORDCORE pRecord ;
} CNRDRAGINFO ;

typedef CNRDRAGINFO *PCNRDRAGINFO ;

```

As was anticipated, the information contained in the memory area pointed at by pRecord refers to the object underlying the mouse's hot spot. If the mouse pointer is over an empty area of the container, then the pRecord will be null. This is a condition that must be checked for in order to implement the copying of an object in a container. In fact, dragging an object to an empty portion of a container, and releasing the mouse's right button, is the typical way to create a copy of the dragged object. The designer must decide what behavior to implement in the application according to the specific development model adopted. In the case of WPSFLDR, it will be possible to create an object of type RED, GREEN, or BLUE directly at the root level (Figure 12.15), an operation that was not implemented in the original FOLDER version.

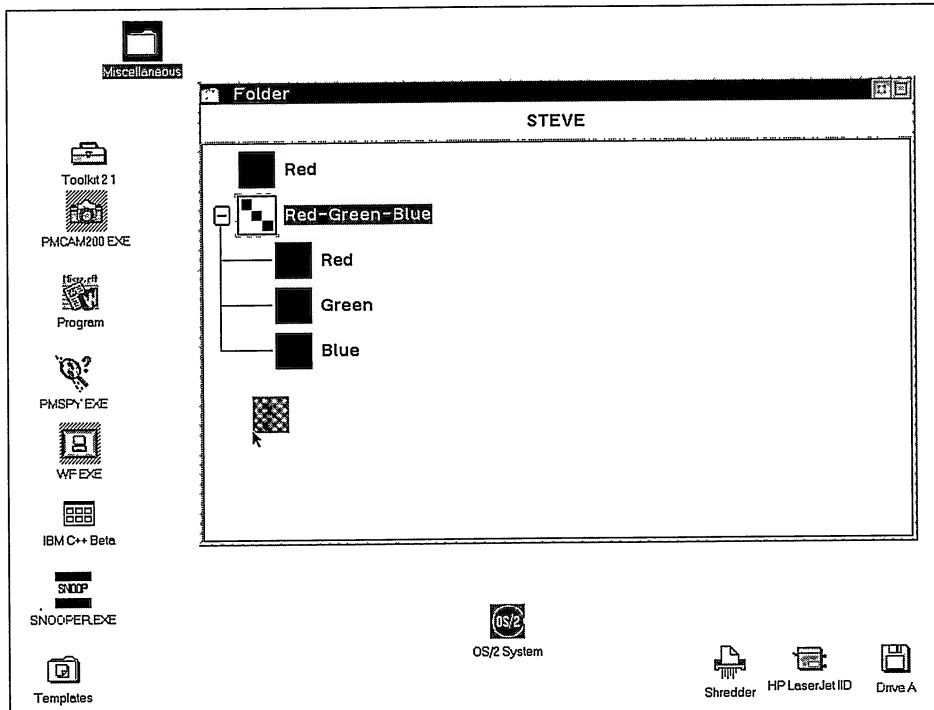


Figure 12.15 WPSFLDR with a red object at the root in the tree display mode.

Before proceeding with an analysis of the listing dealing with the `CN_DROP` notification code, it is important to stress a significant matter which is not documented correctly in the description of `CN_DRAGOVER`. The return value is not reserved; instead it is a combination of two: the `SHORT` containing the `DOR_` and `DO_` defines.

```
...
return MRFROM2SHORT( DOR_DROP, DO_MOVE ) ;
...
```

To avoid the situation shown in Figure 12.15, it is necessary to return the value of `DO_NODROP` whenever the member `pRecord` of the `CNRDRAGINFO` is null:

```
...
// forbid dropping R-G-B objects on the container
if( !prec)
    return MRFROM2SHORT( DOR_NODROP, 0x0000 ) ;
...
```

In the code fragment handling `CN_DROP` you only need to call the *CreateAnotherObject()* function to create new objects on the basis of the direct manipulation performed by the user (Figure 12.16). The source code of WPSFLDR is presented in Listing 12.4.

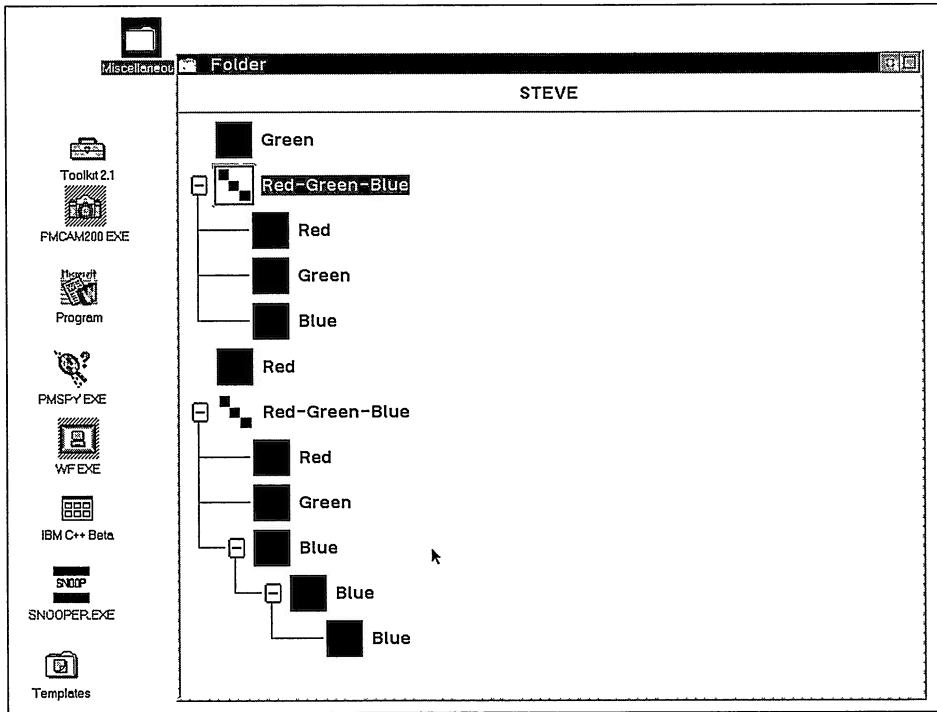


Figure 12.16 WPSFLDR, with its rich contents.

# WPS Programming

If you have read this far, you must be a great OS/2 fan, and are probably eager to make your own hot applications! The purpose of this chapter is, in the first place, to get a better comprehension of the look-and-feel rules characterizing OS/2 2.1 programs; then write a program complying with these rules. We have already covered in many places the style issues of programs. Summarizing the basic rules, we might say: few menu bars, few dialog windows, never a system menu, always a titlebar menu and a window context menu, many objects, lots of drag & drop, care about details, no MDI, always SDI.

This summary might be *too* condensed. Let's expand on each concept by looking at some examples and making some thoughtful considerations. First of all, a question comes to mind. What really is WPS, and what does an *object-oriented* (as the IBM ads say) user interface mean? Take your time, and think about it! WPS is an OS/2 application—a system shell—that mainly takes advantage of the API *Win* services, but also of the *Dos* services for accessing the file system, managing memory, handling multithreading, and more. Furthermore, WPS uses the SOM (System Object Model) objects of OS/2. From a technical point of view, WPS is nothing more than a window of the class `WC_CONTAINER`, which extends over the whole screen area (its frame is hidden behind it). The objects contained in WPS, which we are accustomed to thinking of in four groups (folders, programs, data files, and physical devices), are nothing more than simple records in a container (WPS). A double click on any part of the image will send the notification code `CN_ENTER` to the frame, with all record related information in `mp1`. As we have seen, the data structure `RECORDCORE` of each object is almost invariably associated with some specific memory area, allocated and managed by the `WC_CONTAINER` window designer. So, when a double click affects a record corresponding to an executable, that executable is activated. This also extends to the representation of the other object groups. Actually, object manipulation is handled at the level of a single application, `PMSHELL`, that is WPS, even if it is a very complex application. The drag & drop operations within a container are not at all complex, as we have seen in Listing 12.4. The direct consequence of these considerations is the extreme importance of the class `WC_CONTAINER`, which is a critical and central element when writing code.

The objectives to aim for when designing a new OS/2 application are:

- Creating a product that is strongly integrated in WPS
- The inner workings of the program must comply with the style and operative rules of WPS

---

## Integrating in WPS

You can integrate OS/2 applications into WPS at various levels. After all, even old executables like E.EXE work fine under OS/2 2.1 without creating any great problems. In general, though, there are three successive integration levels in WPS, which are known as minimal, medium, and advanced.

### *Minimal Integration*

Minimal integration is achievable with small modifications to 16-bit code, which can be considered code fixes. An image that better qualifies the features of the product substitutes the system menu icon, but the system menu is not taken away or replaced. The program will have a menu bar, and, consequently, a number of dialog windows displayed after the selection of menu items of the extended command type. An additional improvement is the ability to define a resource of the ASSOCTABLE type in the application's resource file. This integration allows you to obtain the same result that in version 1.x was obtained through associations in the File Managers. However, the association does not involve the placement of a template in the Templates folder, and therefore limits the connection to the file system level. The presence of a template, indeed, involves the user directly and implies a more accurate redesign of the code (for instance, for supporting drag & drop operations).

### *Medium Integration*

The elements that characterize an application of this type are numerous: the adoption of a titlebar menu in place of the old system menu, the presence of the *window context menu*, support of titlebar dragging, and acceptance of other objects dragged by the user. Furthermore, printing is supported directly by dragging the generated objects over the printer icon. The generated documents will have icons of their own, and in the Templates folder there will be a template to allow you to easily produce new ones. This kind of program also uses objects provided by SOM and employed by WPS, like the Color Palette and the Font Palette. These programs are written in C and do not have to resort to the definition of new SOM objects or to subclassing existing ones.

### *Advanced Integration*

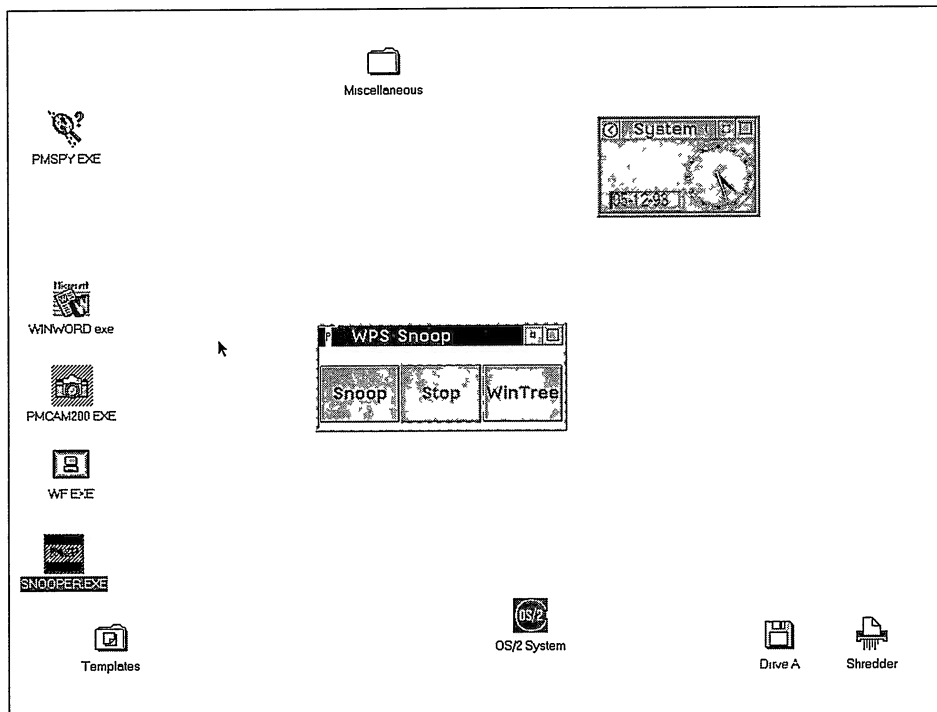
This is the category of "true" OS/2 application. Programs of this category have been redesigned, and have as their design objective the production of something that really interacts with WPS and is truly inspired by the function and style rules of OS/2's user interface. In this case, software development starts with the definition of new objects

at the SOM level, objects that are all integrated with C code. In addition to having all elements described for the preceding categories, these programs take advantage of and rely upon system INI files and on the internal structure of WPS objects for performing a number of operations. Many of the methods of the standard objects, like a printer, are redefined for better adaptation to the specific needs of the program.

## How to Develop for OS/2

The preceding scheme might not be satisfactory in all cases, and is probably excessively rigid in some aspects. The purpose of the above classification is stressing the direction that should be taken when writing OS/2 programs. As we will see in this chapter, it is not at all difficult or demanding to achieve these results. Let's start with a concrete example, an application written in C and integrated into WPS (at the second level, according to the previous classification).

We will now examine SNOOPWSP, a version of SNOOPER that has been revised in some of its functionalities, especially in its user interface: We will take advantage of all techniques learned up to this point. The program identifies itself with a tiny window, positioned at the center of the screen (Figure 13.1)



**Figure 13.1** The look of SNOOPWSP is very different from that of its earlier versions.

The three gray buttons at the center of the *client* immediately capture the user's attention. The first one, Snoop, allows you to activate the *snooping* operations. The logic behind the button pressure consists in the mouse capture and in the subsequent creation of a panel window to display all the related information (it's self-evident that SNOOPWPS provides enough space in its window for displaying information about a window). The central button, Stop, is initially disabled, and eventually terminates any active snooping action. The third and last button, WinTree, introduces some new functionalities with respect to those offered by SNOOPER. We will examine the outcome and the effects generated by the three buttons later. For the moment, let's consider the structural elements of the program. Figure 13.2 emphasizes the presence of the titlebar menu: The overall number of windows in SNOOPWPS is eight (the five traditional windows plus the three buttons).

By pressing the right mouse button in any empty space in the client window, you will display the *window context menu* (Figure 13.3). The interaction between the program and the user follows the traditional scheme adopted by WPS.

One problem that had to be overcome when version 2.0 of OS/2 was released was that of fully understanding the nature of the system shell. SNOOPER helped in this; but it was also necessary to have some means for correlating the information of several windows related by parenthood or ownership. One course of action was extracting from the actual program the data packet retrieved when the mouse's cursor was

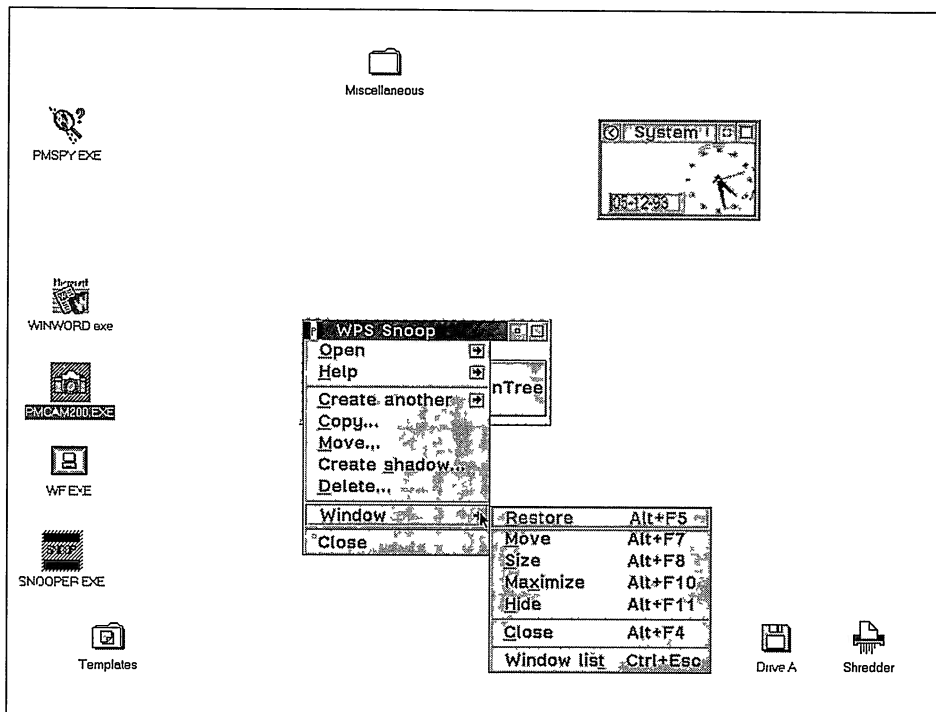
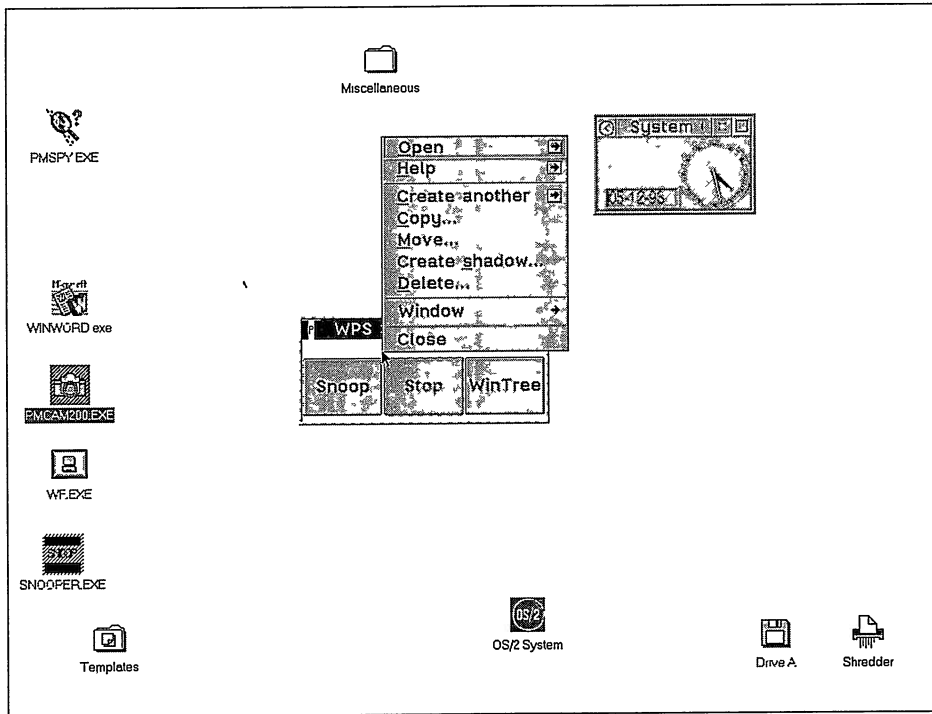


Figure 13.2 SNOOPWPS is equipped with a titlebar menu.

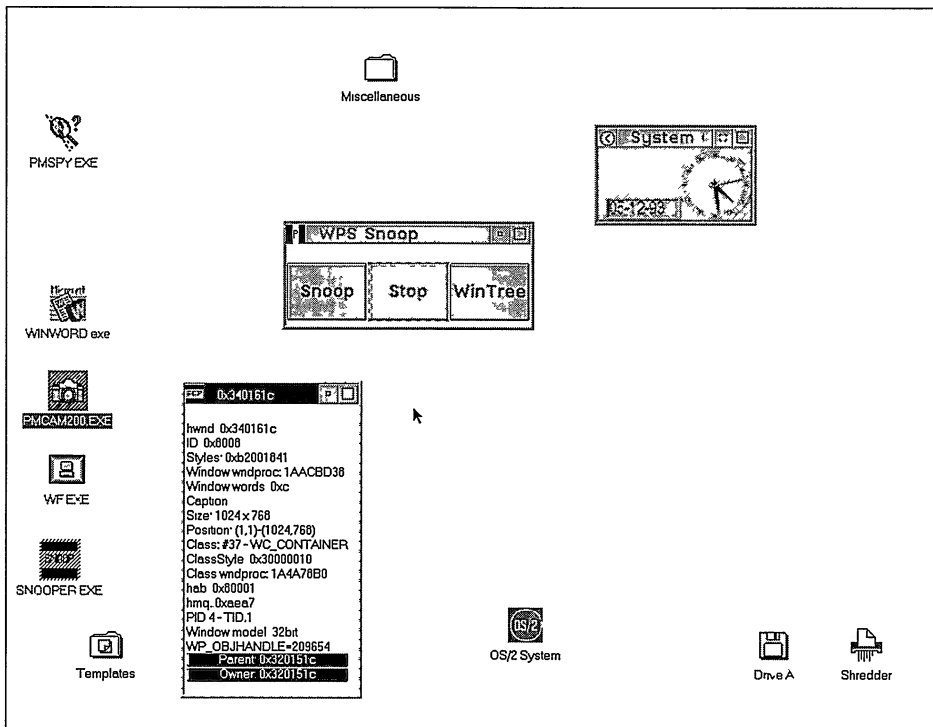




**Figure 13.3** Appearance of the window context menu in SNOOPWPS.

positioned on a new window and storing that data in a separate document. The fundamental idea was permanently logging that data, so that it would be available later for further manipulation. If you click on the Snoop button in the window, nothing special happens: even the cursor's icon doesn't change. From this moment on, though, you are actually spying on the system's activities. The eight windows that make up the program cannot be investigated (this is a design choice). As soon as you reach some other window, a document (a panel) will appear close to the lower left-hand corner of the screen, as shown in Figure 13.4 (it is helpful to have at least SVGA resolution to avoid overcrowding the screen with windows).

The selected font is very small, in order to accommodate a large amount of data in a small space. The output has also been reorganized with respect to the earlier versions of SNOOPER, in addition to presenting a richer data set. SNOOPWPS will also indicate the address of the window procedure stored in the reserved memory is of a window as well as at the class level. Different numeric values indicate that there is *subclassing*. Furthermore, each window specifies its programming model, 16 bit or 32 bit, the position, the size, the PID, and the TID. Each of these data are contained in a window of the class WC\_STATIC. The panel also contains two buttons labeled with the handle of the parent window and of the owner window. You can terminate the snooping actions by pressing the Stop button, which has been enabled, and then



**Figure 13.4** SNOOPWPS shows information about the container window of WPS in a document window named with its handle.

examine in detail the generated document. First of all, both buttons are enabled only if they contain a valid handle—if the window really has a parent and/or an owner. To discover more about them, press the corresponding button (Figure 13.5).

These are stand-alone windows, which the user can position anywhere on the screen. While inspecting and exploring the collected data, you can also look at those windows at the same time. By generating a panel for each investigated window, the screen gets overcrowded easily. The application supports the selective destruction of each data window acting on the titlebar menu or on the special window context menu in the Snoop button (Figure 13.6).

SNOOPWPS will “auto-censor” itself if the user were to request information about an already *snooped* window. This behavior is a consequence of the internal setup of the application, and also affects any possible pressing of the parent and owner buttons inside the panel.

You will often need to take notes about the data produced by snooping. Therefore, a special print option has been devised for printing. Each single document also supports *titlebar dragging* to many destinations: any folder of the interface or even icons of printing devices (Figure 13.7).

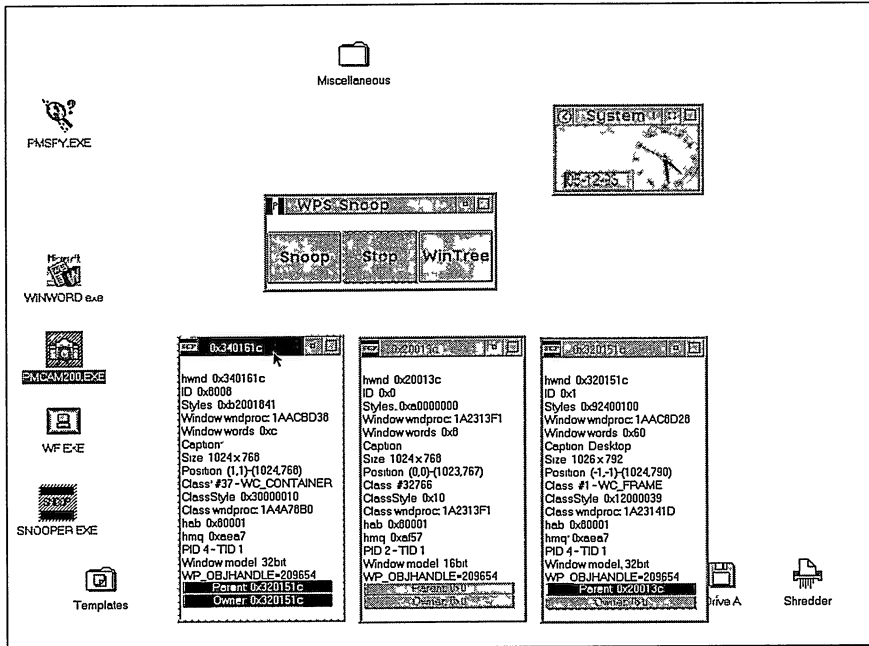


Figure 13.5 Panels showing information about the parent window and the owner window.

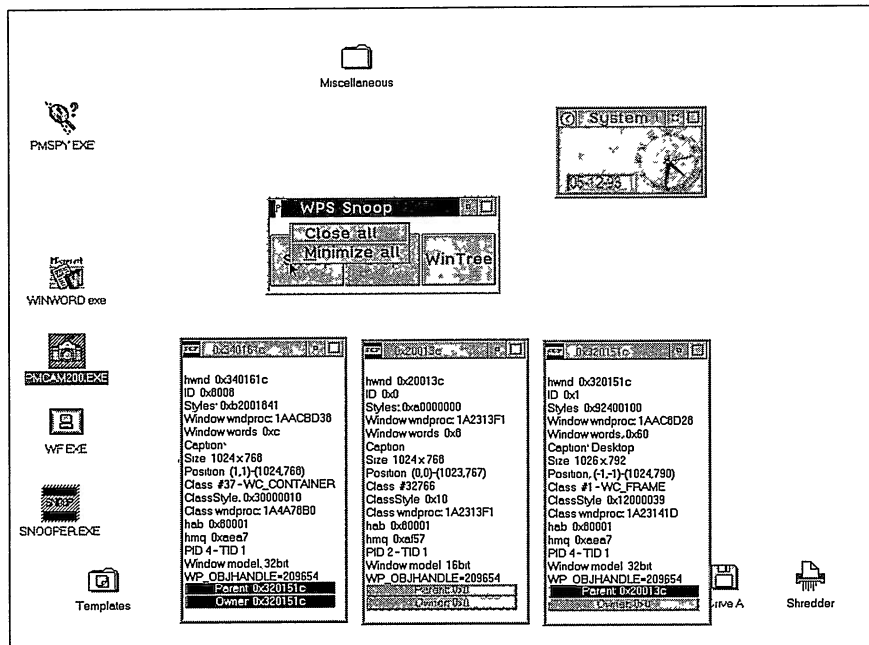
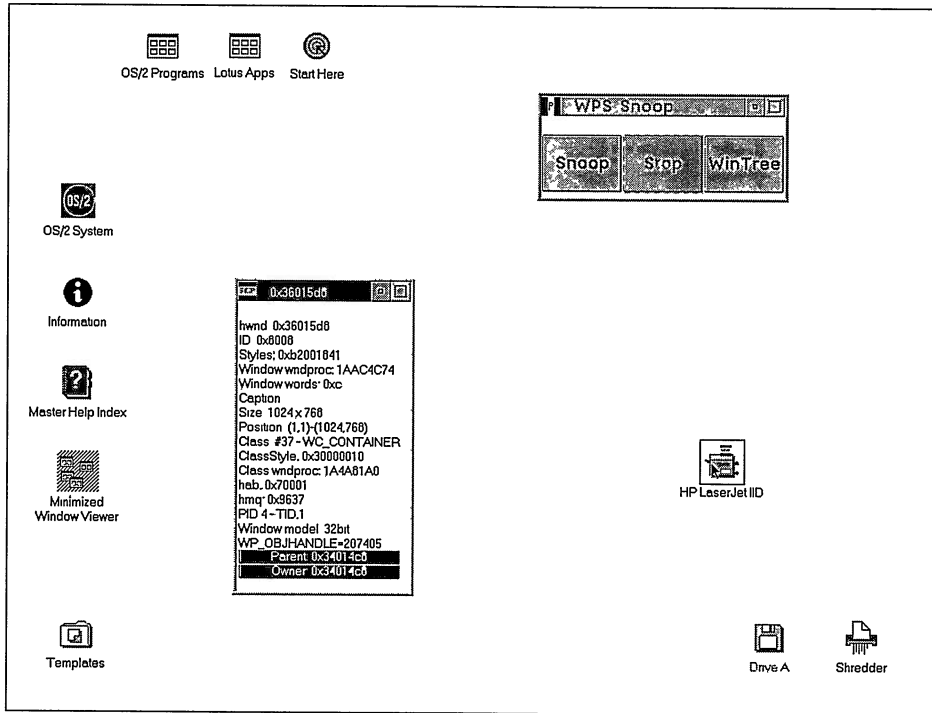


Figure 13.6 Interaction with the data panels.



**Figure 13.7** Printing of the information obtained through SNOOPWPS is achieved by dragging the titlebar icon over the image of a printer.

Another limitation of SNOOPER was the impossibility of examining the features of any nonvisible window or belonging to the tree descending from `HWND_OBJECT`. Any search with the mouse could not solve the problem because `HWND_OBJECT` descendants are always invisible. The only solution to the problem is to explore the entire tree structure of a window, starting from its primogenitor, respectively, `HWND_DESKTOP` and `HWND_OBJECT`. By pressing the WinTree button you will see in the upper left-hand side of the screen a window showing the handles of the windows in a tree structure (Figure 13.8).

By default, after pressing WinTree, the program will scan through all windows, starting from `HWND_DESKTOP`. The choice can be changed by acting on the window context menu of the WinTree button. That's right! Even the buttons in SNOOPWPS, being high-level interaction user interface objects, have a menu of their own (Figure 13.9).

This is a *conditional menu* having `HWND_DESKTOP` as its default. Once the selection has been changed, you can simply press the mouse button over the Default tree menu item to see a new window appear in the upper left-hand side of the screen. A conditional menu always presents a checked menu item. If the user selects the pop-up menu item that introduces a conditional menu, the checked menu item is selected. To

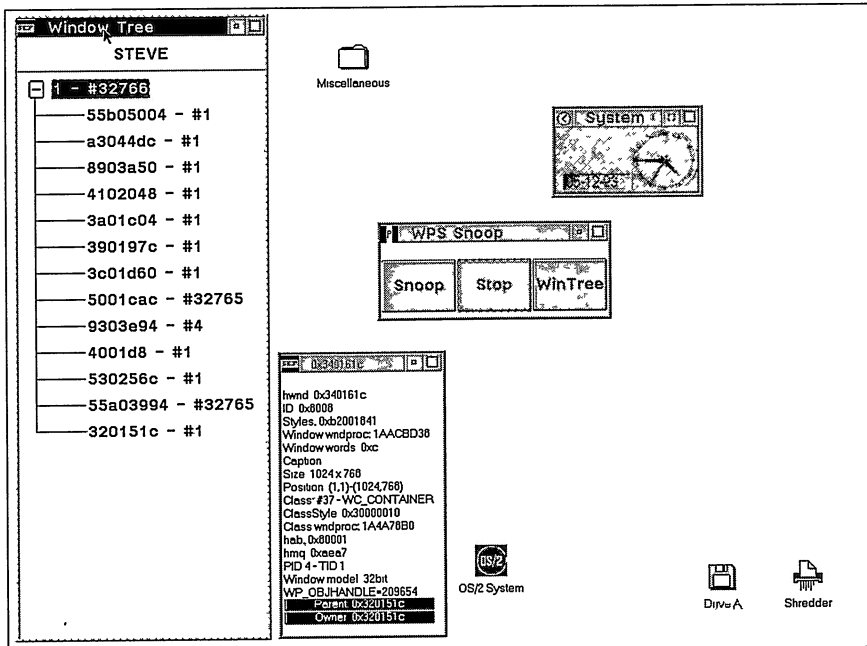


Figure 13.8 The outcome of pressing the WinTree button.

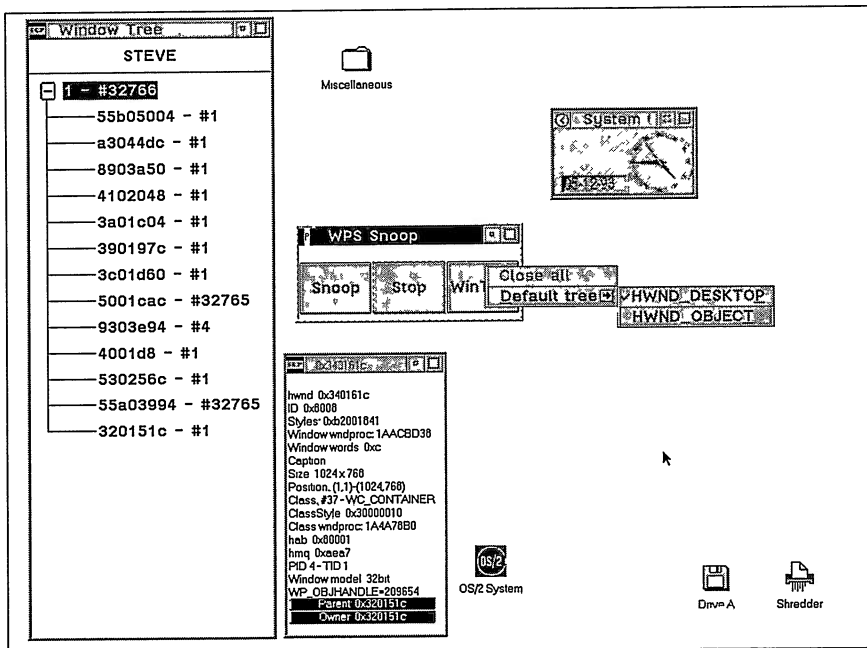


Figure 13.9 The selection of HWND\_OBJECT in the window context menu of WinTree.

set and query the default menu item, you related to the two messages MM\_SETDEFAULTITEMID and MM\_QUERYDEFAULTITEMID.

MM_SETDEFAULTITEMID	0x0432	<i>Description</i>
mp1	ULONGulDefItemID	Default item ID
mp2	ULONG ulRes	Reserved
Return Value	BOOL fResult	Success or failure
MM_QUERYDEFAULTITEMID	0x0431	<i>Description</i>
mp1	ULONG ulRes	Reserved
mp2	ULONG ulRes	Reserved
Return Value	ULONGulDefItemID	Default item ID

As you can see in Figure 13.8, the window tree structure is limited to its first level (first generation descendants). Among these, there are some that are not visible (for instance the *frame* of WPS), while all children of `HWND_OBJECT` are visible by definition. To display the tree structure starting from a generic window, you only need to select that window by dragging it over the WinTree button (Figure 13.10).

Releasing the right mouse button over WinTree displays a new window (Figure 13.11).

The same logic is true also for obtaining the information panel. The only variation in this case concerns the destination button, Snoop (Figure 13.12).

You will probably agree that SNOOPWPS is a significant improvement over SNOOPER. Even with the limited functionality it offers, and the simplicity of the examples, you can certainly conclude that it is a completely different breed of application with respect to its predecessors: It is easier to use, more intuitive, and just better. The ingredients of SNOOPWPS are those you already know: titlebar menu, window context menu, and lots of drag & drop between various components of the program as well as with WPS.

## *Analysis of the Program*

SNOOPWPS is just a framework for a programmer's utility. There are many more functions that might be added, and it can be made much more flexible. For instance, it would be nice to have a *notebook* for its settings, and allow the user to customize the look of the panel windows (maybe by selecting a different system font, colors, and so on), and to customize the program behavior (ignore spying frame windows, for instance). You can create these extensions yourself! It will be a great exercise.

The application registers two window classes, one for the program and the other for the panel windows. The first of the two classes has a total of eight bytes of window words, the sum of a `PVOID` and a `ULONG`. The first four bytes (from 0 to 3) will accommodate a pointer to a memory block dynamically allocated when the message `WM_CREATE` is detected. The block is a page of 4096 bytes; this means that the

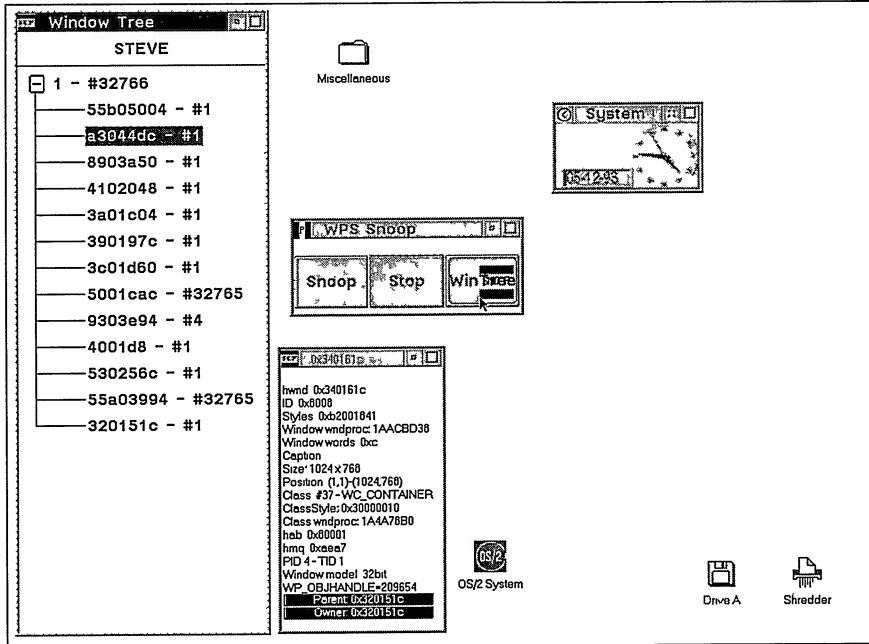


Figure 13.10 Dragging a window handle to the WinTree button will originate a new tree structure containing information about its child windows.

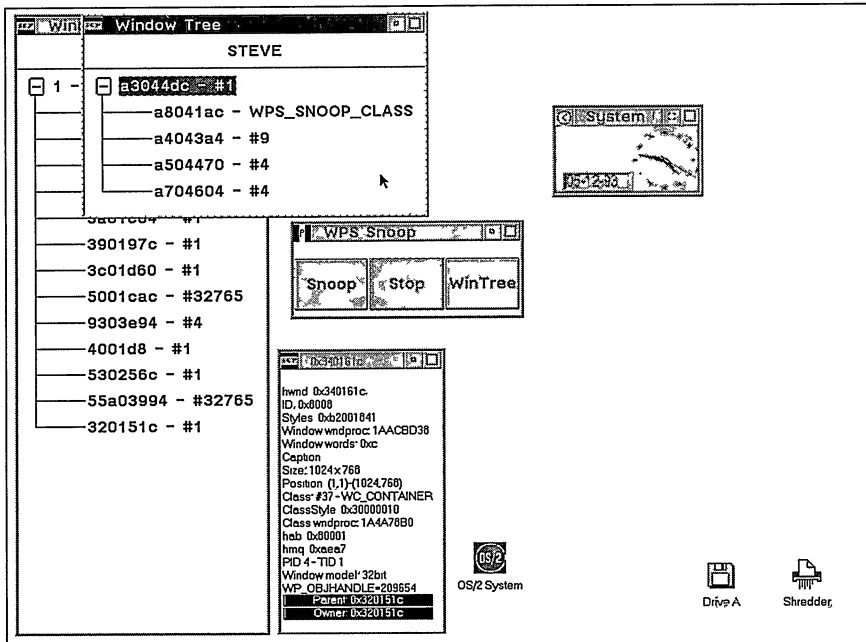
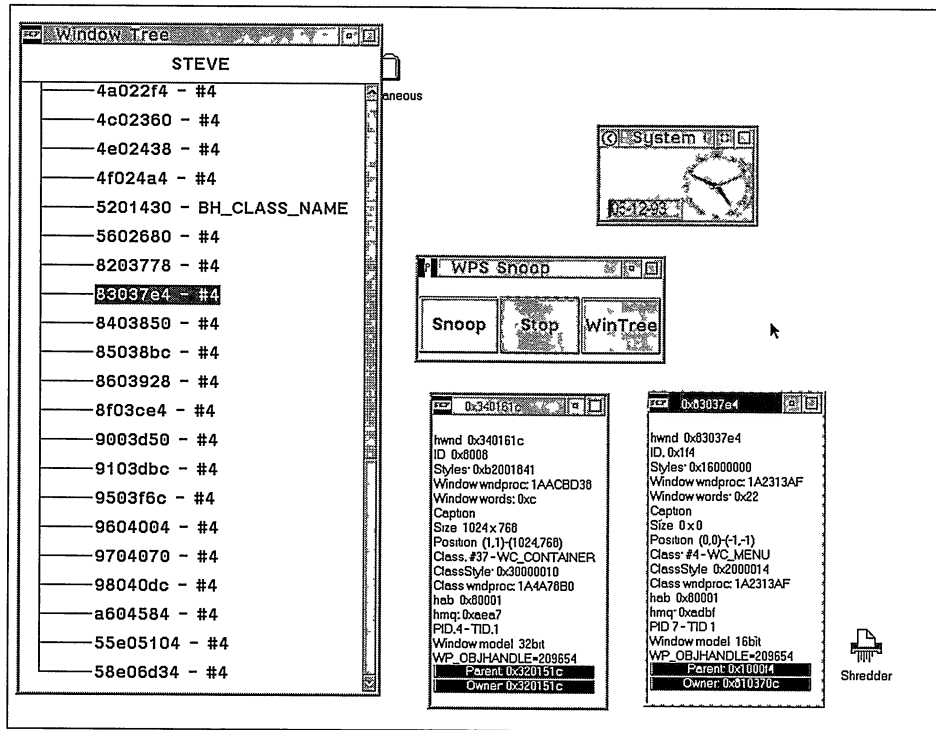


Figure 13.11 The tree structure of the child windows of the handle previously selected and dragged to WinTree.



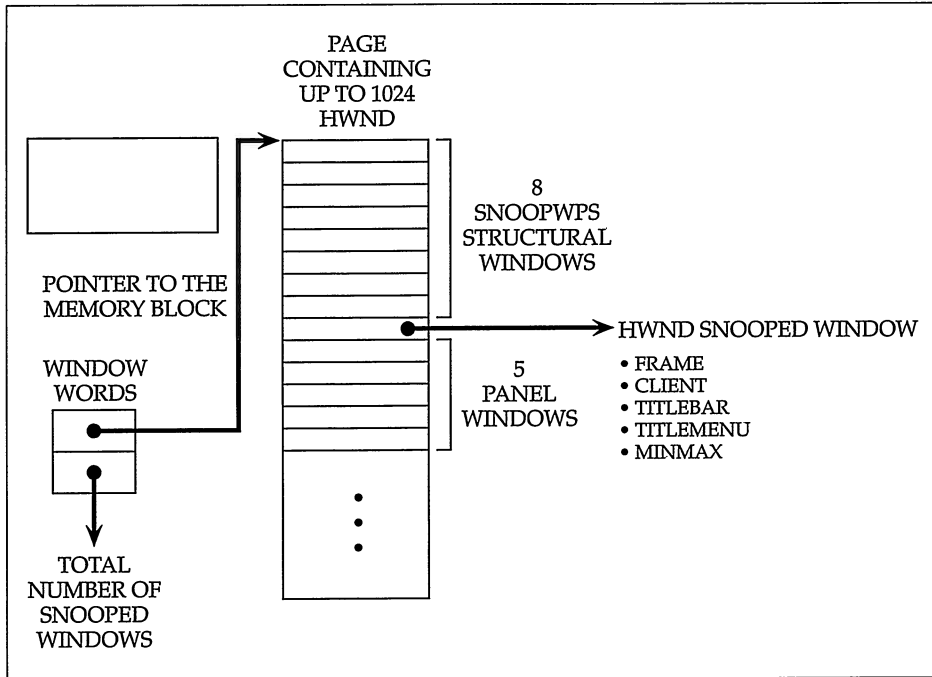
**Figure 13.12** The information panel about a window belonging to the class `WC_MENU`, a drop-down that is invisible to the user.

maximum number of *snoopable* windows sequentially is limited to 1024 ( $1024 \times \text{sizeof}(\text{HWND})$ ). Naturally, you can increase this limit at will without causing any problems by allocating a greater number of consecutive pages. The second part of the window words contains a counter that refers to the overall number of windows handled by SNOOPWPS. This quantity counts both the windows that have actually been examined, as well as the support windows—the eight windows of the main window plus the five windows that make up each panel. When SNOOPWPS is first displayed, only the first eight window handles of the 1024 available in the reserved memory block are actually engaged (Figure 13.13).

When snooping a window, that quantity will be increased by six: the five structural windows of the panel plus the window selected with the mouse. The memory block is filled in sequentially without resorting to any kind of memory management algorithm (linked list, reference series, last recently used, etc.) The simplicity of the data being handled suggests this approach, which is in any case fast and efficient.

As the user closes an information panel, the application executes a clean-up sequence destroying the six involved handles, and making this packet of slots available for any other subsequent snooping operation.





**Figure 13.13** Scheme for managing information in SNOOPWPS.

```

...
case WM_CLOSE:
{
    HWND hwndClient ;
    USHORT usID ;
    PHWND phwnd ;

    // retrieve the owner, the application client window
    hwndClient = OWNER( PAPA( hwnd) ) ;
    // retrieve the pointer to the hwnd list
    phwnd = WinQueryWindowPtr( hwndClient, 0 ) ;
    // retrieve the frame window ID
    usID = WinQueryWindowUShort( PAPA( hwnd), QWS_ID ) ;
    // erase the snooped handle
    phwnd += ( usID - INFOWINDOW ) ;
    *phwnd = NULLHANDLE ;

    // destroy the info panel
    WinDestroyWindow( *( phwnd + 1 ) ) ;
}
return (MRESULT)TRUE ;
...

```

The windows of the class WC\_STATIC and the two buttons present in each panel are interdicted from investigation through SNOOPWPS. Their exclusion, however, is not achieved by storing handles, but by evaluating their IDs. All this happens inside the WM\_MOUSEMOVE message:

```

...
case WM_MOUSEMOVE:
{
    POINTL pt ;
    HWND hwndSnoop ;
    PHWND phwnd ;
    ULONG i ;
    ULONG ulCnt ;
    USHORT usID ;

    // skip if the user hasn't pressed the Snoop button yet
    if( !fSnoop)
        break ;

    pt.x = SHORT1FROMMP( mp1) ;
    pt.y = SHORT2FROMMP( mp1) ;

    // map the points for the DESKTOP
    WinMapWindowPoints( hwnd, HWND_DESKTOP, &pt, 1) ;
    // determine the selected window
    hwndSnoop = WinWindowFromPoint( HWND_DESKTOP, &pt, TRUE) ;

    // skip if it is the STOP button
    if( hwndSnoop == CTRL( hwnd, CT_STOP))
    {
        // release the mouse capture
        WinSetCapture( HWND_DESKTOP, NULLHANDLE) ;
        break ;
    }
    // check the memory area and create the info panel
    CheckWindow( hwnd, hwndSnoop) ;
}
break ;
...

```

After carrying out some preliminary operations in the WM\_MOUSEMOVE message, the actual processing of this portion of the program takes place in the *CheckWindow()* designed specifically for this purpose. This choice allows the user to obtain the same effect when a window handle is dragged from a window produced by pressing WinTree over the Snoop button. *CheckWindow()* will always receive the handle of the application's client, and the handle of the window to analyze. The first operation is retrieving the ID of this last window, and comparing it with those assigned to the windows belonging to the classes WC\_STATIC and WC\_BUTTON appearing in the information panel. Of course, this is not the best way for discriminating between windows used by the application and those present in the system. To improve the algorithm,

there are two viable solutions, both quite simple. The first one is retrieving the parent's handle or the client of the panel. This handle is among the six stored in the memory block managed by the application. The second solution requires the user to store the handles of all windows in the area available to SNOOPWPS. You can choose whichever you please (probably the second one is faster).

```

BOOL CheckWindow(   HWND hwndClient,   // application client window
                   HWND hwndSnoop)    // window chosen by the user
{
    LONG i ;
    USHORT usID ;
    PHWND phwnd ;
    ULONG ulCnt ;
    HWND hmenu ;

    // query the window ID
    usID = WinQueryWindowUShort( hwndSnoop, QWS_ID) ;
    // skip if it is a window inside an info panel
    if( ( usID >= DISPLAYWINDOW) &&
        ( usID <= DISPLAYWINDOW + WINDOW_INFO + 2))
        return FALSE ;
    ...
}

```

To back up through the list of snooped windows or those belonging to the application, you must be able to access the contents of the window words in the program's client, and then loop through the hwnd list quickly.

```

...
// retrieve the pointer to the hwnd list
phwnd = (PHWND)WinQueryWindowPtr( hwndClient, 0) ;
// get the number of window in the block memory area
ulCnt = WinQueryWindowULong( hwndClient, sizeof( PHWND)) ;

// skip if we are not on a different window or on the desktop
for( i = 0; i < ulCnt; )
{
    if( !*phwnd)
    {
        phwnd += SNOOPNEXT ;
        i += SNOOPNEXT ;
        continue ;
    }

    if( hwndSnoop == ((HWND)*phwnd++))
        return FALSE ;

    i++ ;
}
...

```

The handle of the window context menu of the buttons in SNOOPWPS are stored in the four byte window words (QWL\_USER) of the two windows belonging to the class WC\_BUTTON. The menu options of the Snoop button allow the user to destroy or minimize all displayed panels. Both options must be enabled as soon as a new window is examined.

The syntax of *CheckWindow()* is then completed by a call to the function *CreateSnoopWindow()* which physically creates the panel.

```

...
// modify the popup menuitems
hmenu = WinQueryWindowULong( CTRL(hwndClient, CT_SNOOP),
                             QWL_USER) ;
WinSendMessage( hmenu, MM_SETITEMATTR,
                MPFROM2SHORT( MN_CLOSEALL, TRUE),
                MPFROM2SHORT( MIA_DISABLED, ~MIA_DISABLED)) ;
WinSendMessage( hmenu, MM_SETITEMATTR,
                MPFROM2SHORT( MN_MINIMIZEALL, TRUE),
                MPFROM2SHORT( MIA_DISABLED, ~MIA_DISABLED)) ;

// hwndClient  -> main window client window
// hwndSnoop   -> window spotted by the user
return CreateSnoopWindow( hwndClient, hwndSnoop) ;
}

```

## Creating a Panel

The function *CreateSnoopWindow()* retrieves the pertinent data regarding the window selected by the user with the mouse, or indicated through a drag operation; it then displays the data. Let's ignore the logic for filling in the SNOOP which is just slightly more complicated for fulfilling the greater demands of SNOOPWPS; otherwise it is similar to what we have already seen for SNOOPER. The text strings with the description of each piece of information and the values of the members of SNOOP are all contained in one memory block allocated in *CreateSnoopWindow()*.

```

...
// allocate room for the info to output
rc = DosAllocMem( (PPVOID)&pStart, 4096, PAG_COMMIT | PAG_WRITE) ;
if( rc)
{
    WinAlarm( HWND_DESKTOP, WA_ERROR) ;
    WinPostMsg( hwndClient, WM_QUIT, 0L, 0L) ;
}
pchString = pStart ;
...

```

The text strings are separated with a termination character ('\\0') and are all passed to the window procedure of the panel class after the creation of a panel's client.



```

                                INFOWINDOW + u1Pos,
                                (PVOID)&fcd, NULL) ;

// 10.Helv in the titlebar
WinSetPresParam( CTRL( hwndFrame, FID_TITLEBAR),
                 PP_FONTNAMESIZE, sizeof( szFont), szFont) ;

// create the client window. OWNER is the application client window
hwndPanel = WinCreateWindow(   hwndFrame, szSnoopClass,
                                NULL,
                                OL,
                                0, 0, 0, WINDOWCY - 30,
                                hwndFrame, HWND_TOP, FID_CLIENT,
                                pStart, NULL) ;

WinSetWindowPos( hwndFrame, HWND_TOP,
                 ( u1Cnt - SNOOPEROBJ) / SNOOPNEXT * WINDOWCX / 5, 0,
                 WINDOWCX, WINDOWCY,
                 SWP_ZORDER | SWP_SHOW | SWP_MOVE |
                 SWP_SIZE | SWP_ACTIVATE) ;

...

```

The creation of the client will cause, as always, the message WM\_CREATE to be sent to the window procedure of the class to which it belongs. It is in this portion of the code that the text strings are inserted into the WC\_STATIC class window, and into the two buttons identifying the parent and the owner.

The function *WinQueryWindowModel()* indicates the nature of the code that registered the window class by examining an instance of it:

```

#define INCL_WINTHUNKAPI
LONG WINAPI WinQueryWindowModel( HWND hwnd) ;

```

<i>Parameter</i>	<i>Description</i>
hwnd	Handle of the window to be examined
<i>Return Value</i>	<i>Description</i>
LONG	Definition of the programming model employed (16 bit or 32 bit)

The return value is the define PM\_MODEL\_1X or PM\_MODEL\_2X, indicating, respectively, the 16-bit version and the 32-bit version, assuming that the given parameter corresponds to a window handle.

In the code of SNOOPWPS a special text string is stored in the member szModel of the SNOOP structure:

```

...
// query the window model
strcpy( snoop.szModel, (WinQueryWindowModel( hwndSnoop)) ?
        "32bit" : "16bit") ;
...

```

The remaining operation performed in *CreateSnoopWindow()* involves the storage of the given window handle and of the frame handles, with their respective controls, into the memory area managed by the application. The order is not random. The first handle is that of the inspected window, followed by that of the frame. The destruction of a panel is always based on the second one, but even those of global destruction and minimization are driven by the menu of the Snoop button:

```
...
// add the new hwnd and update the pointer
*pMem++ = hwndSnoop ;
*pMem++ = hwndFrame ;
*pMem++ = hwndPanel ;
*pMem++ = CTRL( hwndFrame, FID_TITLEBAR) ; // application titlebar
*pMem++ = CTRL( hwndFrame, FID_SYSMENU) ; // application sysmenu
*pMem++ = CTRL( hwndFrame, FID_MINMAX) ; // application minmax
...
```

*CreateSnoopWindow()* terminates by subclassing the icon of the titlebar menu so that it may be dragged by the user:

```
...
// titlebar subclassing
hwndTitlebar = CTRL( hwndFrame, FID_SYSMENU) ;
WinSendMessage( hwndTitlebar, WM_PASSPROC,
MPFROMP( WinSubclassWindow( hwndTitlebar,
TitlebarWndProc)), 0L) ;
...
```



The source code of SNOOPWPS appears in Listing 13.1. The proliferation of windows on the screen is a consequence of the usage of the SDI model. The information panels are, nonetheless, strictly governed by the application through the Snoop button's menu. The ownership relationship between the application's client and the panels do not find any tool in the API that helps to identify them, as is the case of parenthood. SNOOPWPS solves this problem by storing the frame's handle in the private memory area.

## The API for WPS Objects

We will now explore new territory in OS/2 land. The header file PMWP.H contains the prototypes of fifteen functions (Table 13.1) that are notably different from the traditional tools for managing windows. We have already discussed some of the new functions in Chapter 4, with the expressiveness and ease of use of *WinStoreWindowPos()* and *WinRestoreWindowPos()*. The storage of the position of a window, and even of the presentation parameters, is a distinctive feature of the WPS interface. Furthermore, in Chapter 12 the PANEL application used *WinLoadFileIcon()* to retrieve the icon associated with an object present in the system's interface. Let's concentrate on the API of PMWP.H

**Table 13.1 The PM API Functions for Using the WPS Objects (SOM)**

<i>Function</i>	<i>Description</i>
WinRegisterObjectClass	Registers a new class of objects.
WinDeregisterObjectClass	Cancels a preceding registration of a class of objects.
WinReplaceObjectClass	Replaces a class of object.
WinEnumObjectClasses	Enumerates the classes of objects.
WinCreateObject	Creates an object belonging to a class.
WinSetObjectData	Sets the structural and aesthetic features of an object.
WinDestroyObject	Destroys an object.
WinQueryObject	Returns the handle of an object.
WinSetFileIcon	Sets the icon of an object.
WinFreeFileIcon	Releases the memory associated with the icon of an object.
WinLoadFileIcon	Retrieves the icon associated with an object.
WinStoreWindowPos	Saves the attributes of a window in the system's initialization file.
WinRestoreWindowPos	Retrieves the attributes of a window from the system's initialization file.
WinShutdownSystem	Executes system shutdown.

The group of functions we are about to explore is characterized by the presence of the word *Object*, as in *WinCreateObject()*. This has nothing to do with the *object windows* descending from `HWND_OBJECT`. In this case, it is something completely different. Some folders of WPS, like the System Setup, contain a number of objects that are easy to confuse with true applications. For instance, consider the Color Palette. The overall aspect is that of a generic window containing, probably, a valueset for representing the palette of colors. This is not the case. By examining the Settings window of the Color Palette (Figure 13.14), you can see that this is not an executable module.

One of the components of OS/2 2.1 is known as the System Object Model (SOM), a tool that helps in creating object classes used directly in applications like WPS. The Color Palette is an object of the class `WPColorPalette`, defined in WPS. Therefore, the first test is to get to know the names of the object classes of WPS.



The application `CLASSES` shown in Listing 13.2 will display in a listbox the names of all object classes registered in WPS at any given moment (Figure 13.15). The list can change in time, generally due to the addition of new classes created by programmers. Destroying one of the classes of WPS is an operation to be avoided unless you have some new class with the same functionality.

The logic of this code is based on the function *WinEnumObjectClasses()*, which has the following syntax:



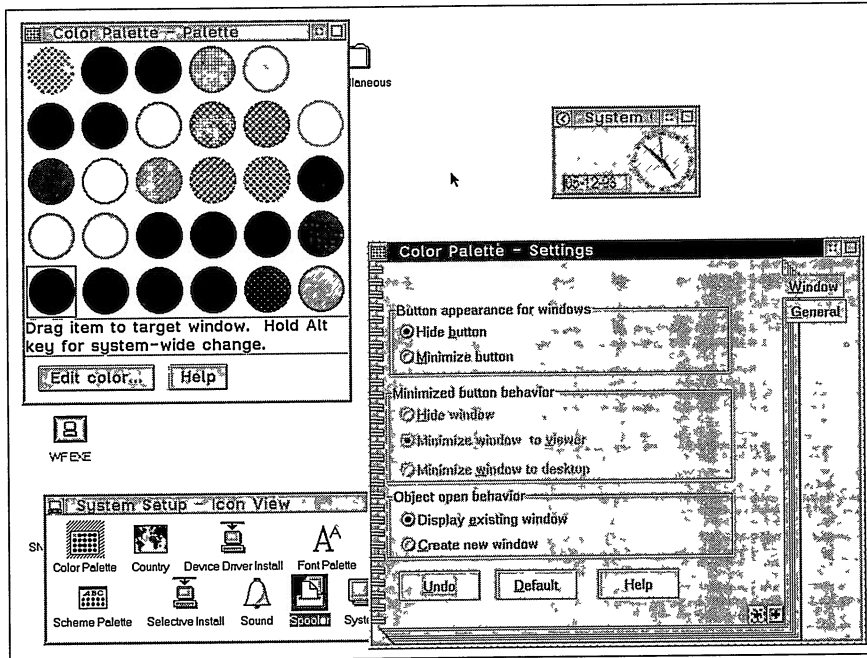


Figure 13.14 The Color Palette is a WPS object, not an application.

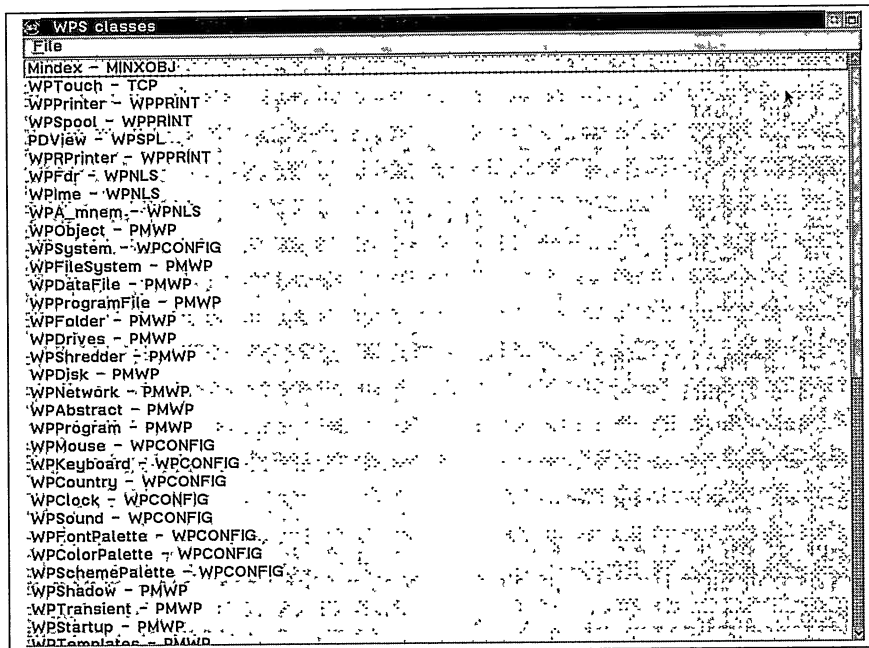


Figure 13.15 The output generated by the application CLASSES.

```
#define INCL_WPCLASS
BOOL WINAPI WinEnumObjectClasses( POBJCLASS pObjClass,
                                  PULONG pu1Size)
```

<i>Parameter</i>	<i>Description</i>
<code>pObjClass</code>	Address of the memory block wherein you can find OBJCLASS structures containing information about the WPS classes
<code>pu1Size</code>	When the function is called, must contain the size of the memory block containing the OBJCLASS structures. When the function returns, contains the actual size of the OBJCLASS structures
<i>Return Value</i>	<i>Description</i>
<code>HOBJECT</code>	Handle of the created object or <code>NULLHANDLE</code> in case of failure

The best way for using this function is to call it once, and specify `NULL` in place of the parameter of type `POBJCLASS`. In this way, the second parameter will report the size of the memory block containing the information describing the classes. The dynamic allocation with `DosAllocMem()` is set in multiples of a page, according to a simple algorithm:

```
...
ULONG u1Size = 0L ;

// get the buffer dimensions
WinEnumObjectClasses( NULL, &u1Size) ;

// allocate the pages
DosAllocMem( (PPVOID)&pObjClasses, 4096 * ( u1Size / 4096 + 1),
             PAG_READ | PAG_WRITE | PAG_COMMIT) ;

// get the classes info
WinEnumObjectClasses( (POBJCLASS)pObjClasses, &u1Size) ;
...
```

The memory block allocated with `DosAllocMem()` is then occupied by a set of structures of type `OBJCLASS`:

```
typedef struct _OBJCLASS
{ // ocls
  struct _OBJCLASS *pNext ;
  PSZ pszClassName ;
  PSZ pszModName ;
} OBJCLASS ;

typedef OBJCLASS *POBJCLASS ;
```

The name of the class and that of the module (the DLL) that contains it, appear in `OBJCLASS` as pointers. This also implies the presence of a member that points to the next `OBJCLASS` structure in the list. The module `WPCONFIG.DLL` contains, among other things, the classes `WPColorPalette` and `WPFontPalette`—the object `Color Palette` is an instance of the first one.

All objects produced by WPS (the Color Palette, for instance) are accessible from a C listing by creating specific instances with *WinCreateObject()*:

```
#define INCL_WPCCLASS
HOBJECT APIENTRY WinCreateObject( PSZ pszClassName,
                                   PSZ pszTitle,
                                   PSZ pszSetupString,
                                   PSZ pszLocation,
                                   ULONG ulFlags) ;
```

<i>Parameter</i>	<i>Description</i>
pszClassName	Name of the class to which the object belongs
pszTitle	Title given to the object
pszSetupString	String containing the object's initialization instructions
pszLocation	Definition of the location to place the object
ulFlags	Flags for creating the object
<i>Return Value</i>	<i>Description</i>
HOBJECT	Handle of the created object or NULLHANDLE in case of failure

This syntax is not complicated, but it introduces some new concepts. First, let's define an object. The Color Palette is an object, just like the Font Palette, and the icons representing the system's drives. The objects of the WPS classes will manifest themselves on screen as windows, which are virtually indistinguishable from those produced traditionally with *WinCreateStdWindow()*. An object is nothing other than a set of code and data originally defined in SOM. Thanks to the syntax of SOM, you can define the behavior and the basic characteristics of a class of objects by integrating all details through PM API calls. This is the case of WPS's folders and of many other components of the interface. This solution allows you to encapsulate code and data, and makes it very easy to employ and reuse them in applications. Without going into greater detail, suffice it to say that in order to create a WPS object it is only necessary to know the name of the available classes.

The class must be indicated as the first parameter to *WinCreateObject()*, and it can be any of those shown in Figure 13.15 and summarized in Table 13.2.

Any text string can operate as an object's title, and will show up in the traditional titlebar of a window. The initialization string of an object contains a series of directives to be executed, and given according to the following syntax:

```
setting=value;setting=value;setting=value
```

Each object has a specific sequence of setting parameters, defined directly by the class to which it belongs. Considering that the goal is to construct an object, the only required initialization is that of specifying OPEN=DEFAULT (with no space in between!) as the startup string.

The fourth parameter establishes the position in the system where the new object will be placed. The possible choices are the defines listed in Table 13.3

These defines identify other standard components of the WPS interface. The syntax of *WinCreateObject()* is completed by a creation flag that can be selected from those listed in Table 13.4.

**Table 13.2 The Classes of WPS**

<i>Class</i>	<i>Class</i>	<i>Class</i>
<b>WPObject</b>		
<b>WPFileSystem</b>	<b>WPAbstract</b>	<b>WPTransient</b>
WPDataFile	Mindex	WPFILTER
WProgramFile	WPrinter	WPFinder
WPCommandFile	WPRPrinter	WPMinWindow
WPrinterDriver	WSPool	WPCnrView
WPBitmap	WPSystem	WPFolderCV
WPPointer	WPShreder	WPDiskCV
WIcon	WPDisk	
WPMet	WPProgram	
WPPif	WPMouse	
WPFolder	WPKeyboard	
WPDives	WPCountry	
WPNetwork	WPClock	
WPStartup	WPSound	
WPTemplates	WPPalette	
WPDesktop	WPFontPalette	
WPFindFolder	WPColorPalette	
WPRootFolder	WPSchemePalette	
WPNetgrp	WPWinConfig	
WPServer	WPPower	
WPSHaredDir	WPKlist	
WPMinWinViewer	WPSHadow	
	WPNetLink	

**Table 13.3 Some of the Standard Locations to Place Objects in WPS**

<i>Location</i>	<i>Description</i>
<WP_NOWHERE>	In the hidden folder
<WP_DESKTOP>	In the desktop
<WP_OS2SYS>	In the OS2SYS.INI configuration file
<WP_TEMPS>	In the templates' folder
<WP_CONFIG>	In the System Setup folder
<WP_START>	In the Startup folder
<WP_INFO>	In the Information folder
<WP_DRIVES>	In the Drives folder

**Table 13.4 Creation Flags for *WinCreateObject()***

<i>Flag</i>	<i>Value</i>	<i>Description</i>
CO_FAILIFEXISTS	0	The object is not created if there already exists another one with the same ID.
CO_REPLACEIFEXISTS	1	The object replaces another one with the same ID.
CO_UPDATEIFEXISTS	2	The object updates any existing object with the same ID.

The return value is a handle to the object. This is a piece of information used only from within other functions of this category, like *WinSetObjectData()*, but that has nothing to do with the window adopted by the object to appear on the screen. The syntax for obtaining an instance of a Color Palette is the following:

```
...
HOBJECT hobjClr ;
...
hobjClr = WinCreateObject( "WPColorPalette",
                          "Object color",
                          "OPEN=DEFAULT",
                          "<WP_DESKTOP>",
                          CO_UPDATEIFEXISTS))
...

```

Once created, the object will appear on the screen as an ordinary window. The interactions between the user and the object are in practice the same as those between the user and an ordinary window, at least for the standard structural elements: the titlebar, the sizing icons, and so on. The objects present in the System Setup folder are the result of a call to *WinCreateObject()*, without specifying in the initialization string the OPEN directive. A double click on the object will cause the object to be opened in its standard mode.

Do you remember how to create a window of the WC\_CONTAINER class? You can achieve the same result in a much simpler way by calling *WinCreateObject()* with the following syntax:

```
...
HOBJECT hobjClr ;
...
hobjClr = WinCreateObject( "WPFolder",
                          "New folder",
                          "OPEN=DEFAULT",
                          "<WP_DESKTOP>",
                          CO_UPDATEIFEXISTS))
...

```

The result is shown in Figure 13.16, after some objects taken from WPS have been placed in the folder.

By inspecting *New folder* with SNOOPWPS you can see that it is a window of the class WC\_CONTAINER. In practice, creating objects of the classes registered in WPS is an ideal way to simplify the writing of code for OS/2 programs. Inverting the line of reasoning, WPS objects are nothing but large portions of code and data that take advantage of PM's API.

At the current stage there is a limitation in the interaction between applications and WPS objects: the lack of a tool that allows retrieving the handle of the window representing the object. Indeed, when writing C code, the fundamental element of the whole of the designer's work is managing windows through their handles. It is likely that the same is true for objects. It would be useful to have some function that, given the handle of an object (HOBJECT), was able to return the handle of the associated window, and/or information about its status (closed object, minimized, and so on). Currently there is nothing like this, and therefore we have to find a different solution.

## A Simple Installation Program

The extreme ease of creating a folder with *WinCreateObject()* is the leading idea for implementing a very simple program. In WPS, folders are also a visual representation

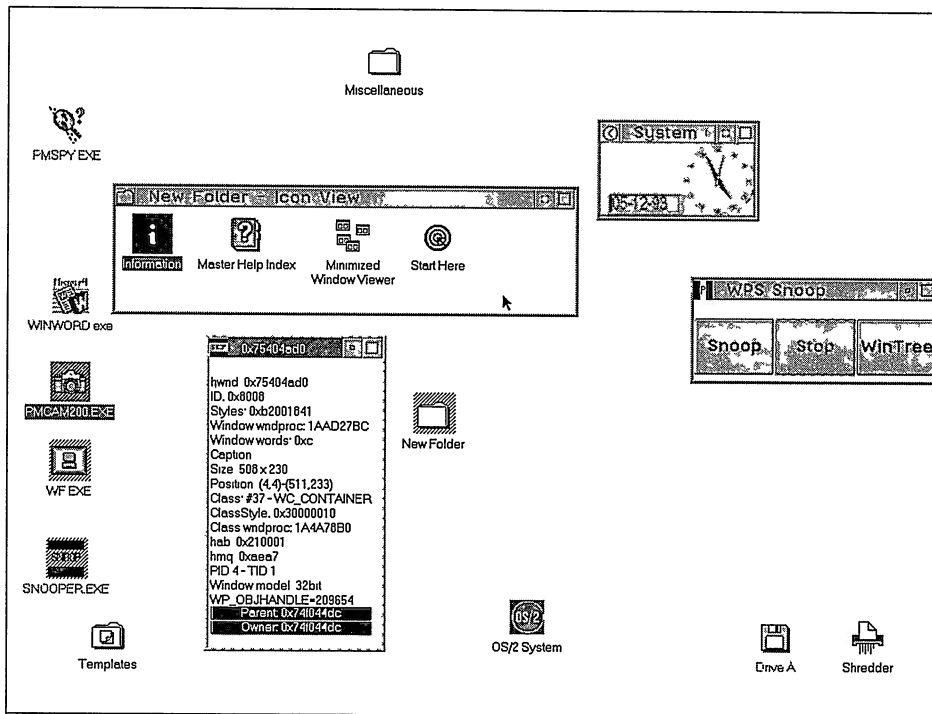


Figure 13.16 A folder created as an object.

of a directory in the file system. The creation of a folder called TWENY in the system's desktop is the equivalent of generating a directory on the bootup disk under the directory \DESKTOP (in OS/2 2.1 the desktop is called simply DESKTOP, and no longer Desktop 210 as in version 2.0). We can take advantage of the relationship between WPS folders and the file system's directories. The installation of an application from floppy disks on the hard disk requires a two-level interaction with the file system:

- Create the directory in which to place the files
- Copy the files

At first glance, these might seem to be operations that could be delegated to the *Dos* API of OS/2. The first one, though, can be replaced by the creation of a folder, displayed in the preferred mode (icon view or detail view, for example), through the specific object API. To create a folder, just call *WinCreateObject()* specifying the *WPFolder* class. To optimize performance it is convenient to create and display it on the screen as an icon. Only after all the files involved have been copied to the target directory do we open the folder calling *WinSetObjectData()* with the appropriate setting string. Separating the creation from the opening is the the best approach in terms of how promptly the object appears on the screen. In the case of the *INSTALL* program, the customization goes a little further. Not only do we create a folder, but we also substitute a blue version of the traditional folder image for the standard icon. Each object class supports some initialization commands in the setup string. One of these is *ICONRESOURCE*, which adopts a specific icon as the object representation on the screen. Its syntax requires the icon ID, followed by the DLL module name where the icon is located. Despite its simplicity, you must follow these two guidelines:

- The icon ID has to be a number, not a definition
- Specify only the DLL name without the extension (.DLL)

The following code fragment shows how to create a new folder on the desktop with a different icon:

```
...
// create the install folder
hobj = WinCreateObject( "WPFolder",
                       "Install Folder",
                       "ICONRESOURCE=301,MIE02",
                       "<WP_DESKTOP>",
                       CO_REPLACEIFEXISTS) ;
...
```

Before copying the files you must understand this crucial piece of information—the bootup disk in a OS/2 2.1 system is no longer limited to the first hard drive, C:. It can be any partition, either primary or logical. *DosCopy()* needs to know the full pathname to where it should copy the files. To get the boot-drive letter, you have to rely on *DosQuerySysInfo()*:

```

...
DosQuerySysInfo( QSV_BOOT_DRIVE, QSV_BOOT_DRIVE, (PVOID)&lDrive,
                 sizeof( lDrive) );
...

```

This function returns one or more pieces of data in its third parameter according to the values of the first two.

```

#define INCL_DOSMISC
APIRET APIENTRY DosQuerySysInfo( ULONG iStart,
                                ULONG iLast,
                                PVOID pBuf,
                                ULONG cbBuf) ;

```

<i>Parameter</i>	<i>Description</i>
iStart	First information to retrieve
iLast	Last information to retrieve
pBuf	Pointer to a buffer to store the requested set of data
cbBuf	Buffer dimension
<i>Return Value</i>	<i>Description</i>
APIRET	Zero if the function operated successfully

To get the boot drive number, both `iStart` and `iLast` are set to `QSV_BOOT_DRIVE`. These values are then transformed into the drive letter to form the full pathname for the Desktop. `DosCopy()` does the filling of the new folder. When the copy operation is over, `INSTALL` (Listing 13.3) calls `WinSetObjectData()` to open the folder showing its contents. The program presents itself on the screen with a listbox as its client, inside which the names of all files present in the current directory are listed (Figure 13.17).

The selection of the Install option from the File menu will first cause the creation of the Install Folder, and then physically copy what is contained in the listbox to the new directory, through `DosCopy()`. The result is shown in Figure 13.18.

The creation of a new folder with `WinCreateObject()` involves much more than the mere production of a window of the `WC_CONTAINER` class. We can take advantage of the fact that a WPS folder corresponds to a directory—and therefore implies the creation of a new directory in `WinCreateObject()` by specifying the class `WPFo1der`—and that it displays its own contents automatically in the `ICON` mode by default. To see what information is actually present in the Details view, you only need to act upon the object's *window context menu* (Figure 13.19).

The code of `INSTALL` also introduces another interesting novelty. As you might suppose for a WPS-compliant application, in this case *titlebar dragging* is supported. The preliminary operations are similar to those described in the `DRAG` example (Listing 13.1): subclassing of the titlebar menu and then interception of the message `WM_BEGINDRAG`. Different from `DRAG`, though, the information transferred with drag & drop refer to a series of files that are physically present on some media (probably a floppy disk). In this case, it is convenient to use the support function `DrgDragFiles()` which simplifies all operations covered by the process.





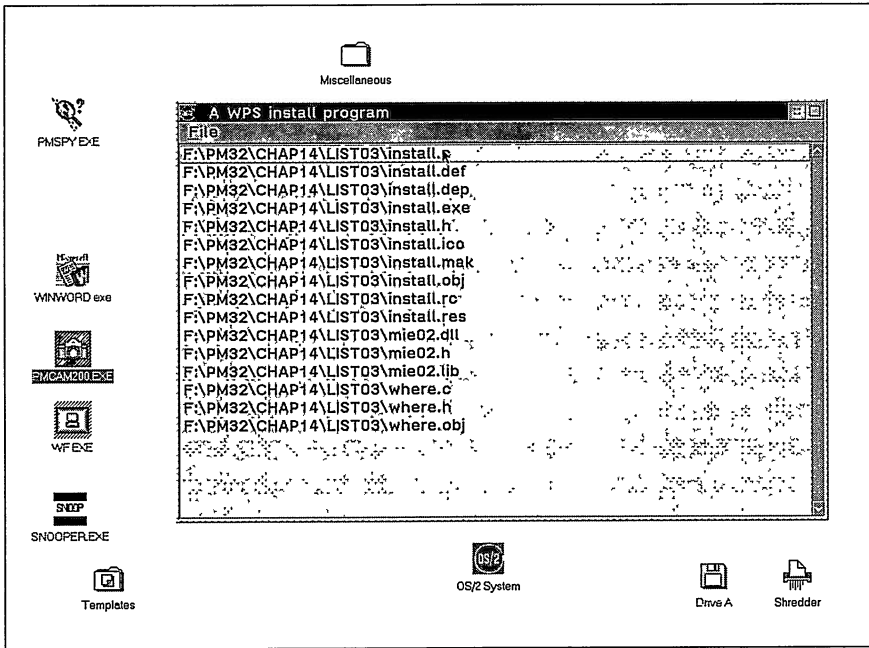


Figure 13.17 The INSTALL immediately after its activation.

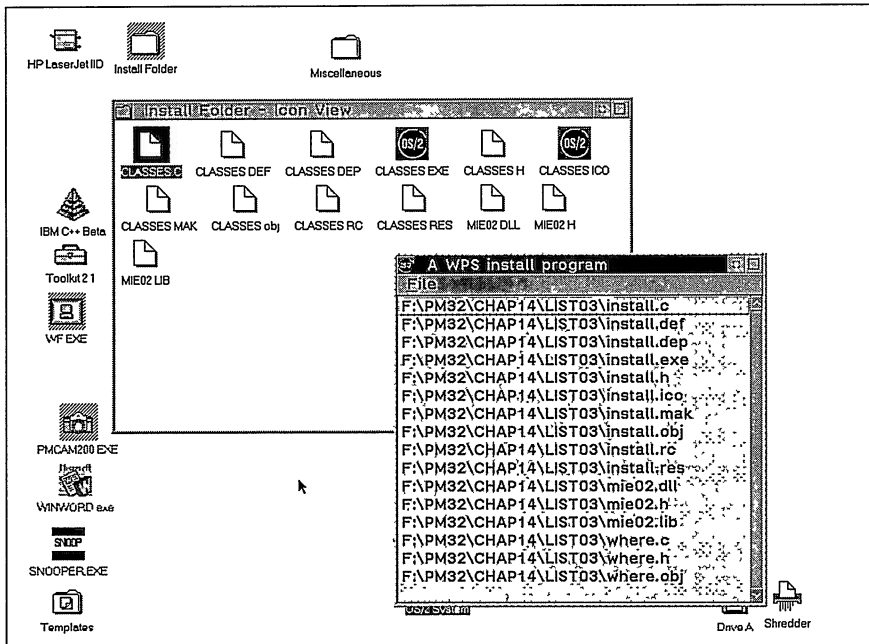


Figure 13.18 Display of the files copied by the INSTALL program to a new folder in the system.

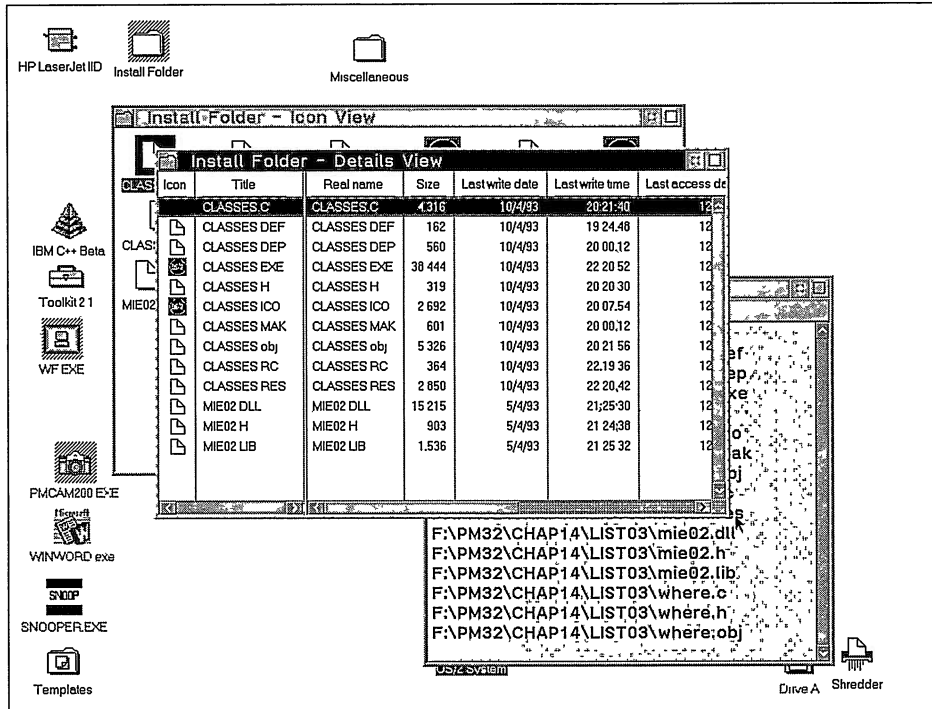


Figure 13.19 The display of the folder Install Program in Details view.

```
#define INCL_WINSTDDRAG
BOOL APIENTRY DrgDragFiles( HWND hwnd,
                             PSZ *apszFiles,
                             PSZ *apszTypes,
                             PSZ *apszTargets,
                             ULONG cFiles,
                             HPOINTER hpPtrDrag,
                             ULONG vkTerm,
                             BOOL fSourceRender,
                             ULONG ulRsvd) ;
```

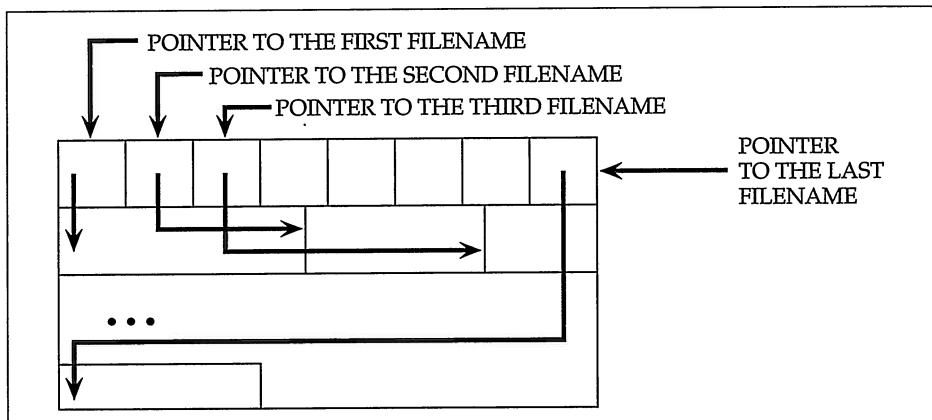
Parameter	Description
hwnd	Handle of the window wherein the drag & drop operations are initiated
apszFiles	Pointer to a CHAR containing the addresses of the file names to be dragged
apszTypes	Pointer to a CHAR containing the addresses of the file types to be dragged
apszTargets	Pointer to a pointer to a char containing the addresses of the names to assign to the files during dragging

cFiles	Number of files being dragged
hptrDrag	Handle of the icon to be displayed during dragging
vkTerm	Virtual key indicating termination of dragging
fSourceRender	Indicates whether the caller will receive a DM_RENDER message for each file
ulRsvd	Reserved
<b>Return Value</b>	<b>Description</b>
BOOL	Success or failure of the operation

The syntax of *DrgDragFiles()* only seems complex. After the handle of the window engaged as the source of the drag & drop operations, there are three parameters, all of the same type (PAPSZ) even if in the declaration they are expressed as pointers to a PSZ. This is, in practice, a pointer to a pointer to a UCHAR. The strategy behind this is very similar to what we have seen in implementing a *spinbutton* containing text strings (Listing 7.13). The application will take care of dynamically allocating the memory block and reserve for itself the first portion for storing the addresses of the file names placed after the last pointer (Figure 13.20).

By indicating NULL for the parameter pTypes you can instruct *DrgDragFiles()* to retrieve the type of each single file affected by the drag & drop operations directly from its extended attributes. This is the most convenient solution. The same value can be given to pTargets. In this way, the dragged files will take on the same name in the target folder as they had in the source folder, since they now have a different pathname. The number of files indicated in the cFiles parameter is used by *DrgDragFiles()* to determine how many memory addresses are needed for handling the file names.

The image associated with the mouse pointer is a simple icon that can be loaded with *WinLoadPointer()*. WPS uses a standard bitmap when more files are being dragged. The application's own icon is the solution adopted in INSTALL. The



**Figure 13.20** The scheme used for defining the second parameter of *DrgDragFiles()*.

parameter `vkTerm` corresponds to the virtual key that indicates the termination of dragging. It is advisable to use the standard solution represented by `VK_BUTTON2`. The syntax of `DrgDragFiles()` is then completed by an indicator, the `fSourceRender` parameter, that establishes if the passing of files is to take place directly under the function's own control (`TRUE`) or by means of some subsequent interaction with the source application (`FALSE`). In `INSTALL` it is more convenient to let `DrgDragFiles()` take care of transferring the files. However, it is very important to specify as the source window handle the titlebar menu. This assures the reception of the message `DM_RENDERCOMPLETE` for each file that reaches the final destination correctly. Furthermore, the window procedure of the subclassed titlebar menu will receive the message `WM_ENDDRAG`. This is the ideal place to destroy the memory block previously allocated.

Figure 13.21 shows the `INSTALL` program in full action, when the entire contents of the running module's directory are being dragged. The interaction with WPS is total, as you can infer from the presence of the dialog that lists the files being dragged.

`DrgDragFiles()` eliminates the need for structures like `DRAGINFO`, `DRAGITEM`, and `DRAGIMAGE` and a great deal of the preliminary setup work for drag & drop. Releasing the titlebar icon over any destination folder almost invariably corresponds to a *move* operation. To transform this into a *copy* operation, the user must press the `CTRL` key.

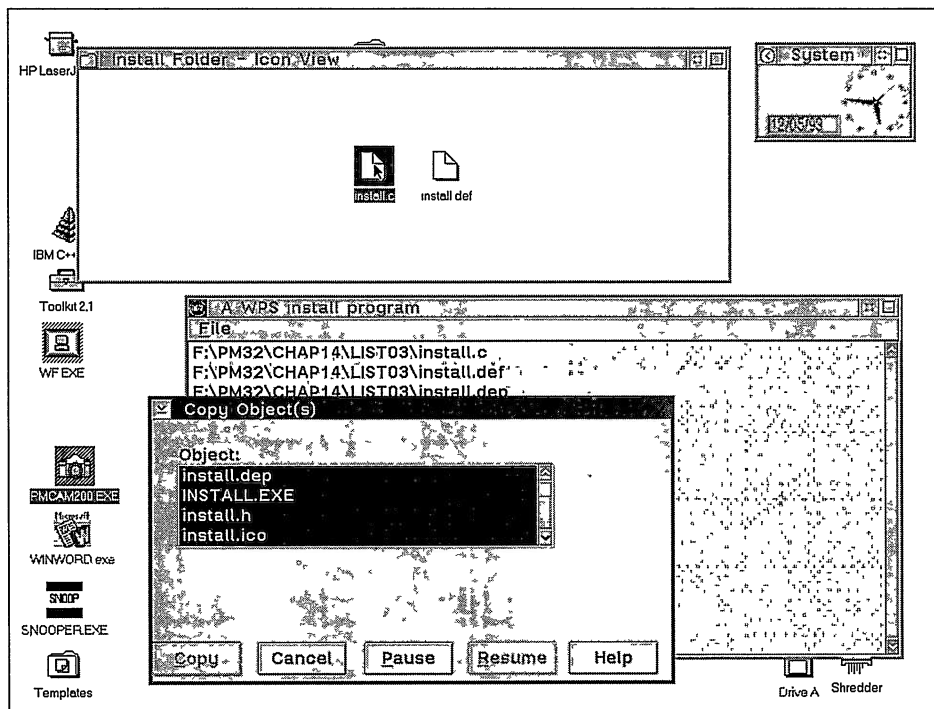


Figure 13.21 Dragging of the files present in the listbox of `INSTALL` by using `DrgDragFiles()`.

## Destroying an Object

The rule that states that each element must be destroyed after its use holds even for objects; this is why the API provides the *WinDestroyObject()* function:

```
#define INCL_WPCCLASS
BOOL APIENTRY WinDestroyObject( HOBJECT hobject) ;
```

<i>Parameter</i>	<i>Description</i>
hobject	Handle of the object to destroy
<i>Return Value</i>	<i>Description</i>
BOOL	Success or failure of the operation

The only parameter of this function is the handle previously returned by *WinCreateObject()*. It follows that this piece of information must be preserved throughout the application's code. Object handles are persistent, as opposed to window handles. Each object always maintains its own handle for all its life, and for any subsequent activation of the whole system. Now, let's move to a more ambitious project.

---

## Developing a WPS Editor

The basic guidelines for the development of a program complying with the WPS style rules are the following:

- No menu bar
- Presence of a *window context menu*
- Use of the titlebar menu
- Heavy use of the objects furnished by WPS
- Implementation of drag & drop operations, within the program, as well as with the system's interface.

You don't have to waste time looking for these functionalities in the Productivity folder. Unfortunately, the applets provided here are not at all instructional for those designers who wish to develop OS/2 programs truly integrated in WPS. However, there are some excellent examples that can demonstrate exactly the opposite—how *not* to develop an application for PM. One of these is the system editor, E.EXE. The least you might expect from an application like this is some printing ability.

Examining E.EXE you can recognize some larger component at the interface level as well as in terms of functionality. A great deal of the application's logic is governed by the adoption of a window of the WC\_MLE class for dealing with the user's input operations. The menu bar presents only four top-levels, and the use of numerous dialog windows, some of which are of the predefined type (including Open, Save, Save As...). Without oversimplifying, the work of the designers of E.EXE has largely been that of assembling various components of PM's API according to the CUA 89 style rules. We can do the same, referring to the CUA 91 specifications and to the

behavioral and aesthetic rules of WPS. Here are the operations that an editor should provide to its users:

- Load a preexisting ASCII file
- Edit a document from scratch
- Save the document being edited whenever the user demands so, and giving it a name as the user pleases
- Perform printing
- Interact with the Clipboard
- Perform search and replace operations
- Change the text's font
- Change the colors of the background and of the text
- Set options, like the displaying the current line number

The last two points are not absolutely mandatory, but nonetheless fall into the set of features commonly supported. On the whole, the above list corresponds to the contents of the drop-down menus of E.EXE. The traditional method of interacting with the user is that of presenting a number of menu items distributed among the various cascading menus. WPS, on the other hand, prescribes interaction according to a different scheme.

## *Loading a File*

This operation is usually performed in two ways: acting upon an object produced with the editor and taking advantage of the document-application association, or by dragging a file directly over the program. If you think about this, both operations are not that much more demanding than the traditional approach; rather they have undoubted advantages. Forcing the user to manage the PC by concentrating on the objects present on the desktop is unquestionably much better than presenting a dialog window containing a number of controls, where the concepts of the file system (directory, drive, and pathname) are always present, requiring specific knowledge. Navigating among folders is a simple operation that anybody can manage without knowing anything about the structure of disks, partitions, and formatting standards. This approach simply emphasizes the *object-oriented* nature present in WPS. In this way, applications are no longer the focal point of all attention; rather they are just tools for handling specific data.

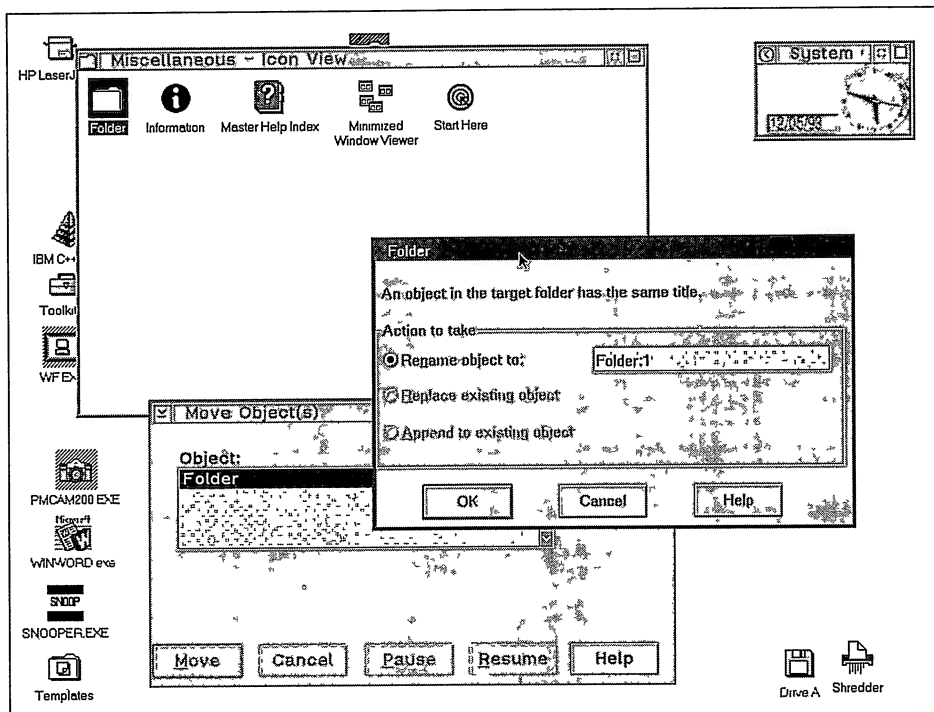
## *Editing a New Document from Scratch*

An editor for OS/2 2.1 must be developed according to the SDI model, with only one input window given to the user. The writing of a new document thereby implies the deletion of the current contents of the editing window or the activation of a new instance of the program. The best approach is to have a tool for eliminating the existing text in only one user interaction. The *window context menu* of the program will therefore

present the menu item New. Another alternative is to select the Create another menu item in the window context menu to place a new instance of the application on the desktop (like a new object) or to open a new executing copy of the program.

## Saving a Document

Traditional applications always have the Save and Save As... options for allowing the user to save the current document. A simpler and more intuitive way is through *titlebar dragging*, as shown in Chapter 12. Even in this case, the user does not have to know anything about the file system, but only needs to manage the objects present in WPS. The simplest and most straightforward method of saving is to position a document directly in the desktop or in some open folder that might be previously prepared for the saving operation. The implementation of something equivalent to the Save As... option is no longer necessary. What has been said for Save still applies for saving and changing names. When you try to save a document with the name of some existing object, WPS warns you of the problem and pops up a dialog window to ask the user to decide what should be done (Figure 13.22).



**Figure 13.22** The dialog displayed by the system when you try to place an object in a container where something with the same name already exists.

This is an ideal tool for assigning a new name to the object or for overwriting the existing one. You might be skeptical about this new approach, but keep in mind that there is no rule that forces an application to present the menu items Save and Save As... for saving your work in a named file. It is only out of habit that you might think that this is the standard way.

## Printing

With the term *printing* we generally refer to both the actual printing (the Print menu item), and possible changes to the print attributes made by interacting with the printing device driver (the menu item Print setup). The documents produced with an editor are not very sophisticated, as far as typesetting, fonts, and other aspects which are more sophisticated in the realm of word processors or DTP tools. The interface with the print drivers installed in the system can be done through an option of the program's *window context menu*, even if this is not a mandatory requirement. Even regarding the issue of printing there are some special considerations introduced by WPS. The interface will allow a variety of representations (icons) of the same physical output device, associating each one of them with different print attributes (like page orientation, for example). An expert OS/2 user could probably set up various configurations for the printer, and put them in some handy folder. Then, to perform custom printing, the user only needs to drag a document over the appropriate device image, and release the right mouse button.

This operation must be allowed by any application, always by implementing *titlebar dragging*. Printing is thus reduced to the execution of a well-known and easy-to-perform gesture: the dragging of the titlebar icon.

## Interacting with the Clipboard

In discussing the previous points, we have considered stylistic and functional solutions that are quite far away from the habits and rules of CUA 89. In this case, though, the adoption of a secondary menu in the application's window context menu containing the traditional options for cutting and pasting is the best solution. The adoption of accelerators and the well-established habits of users with this sequence of keys is a guarantee of success, and does not affect the look and feel of the program.

## Searching and Replacing Text

At the current stage, WPS does not have any special object for performing these operations. It is a weak point of the interface. The Find... option is present in any WPS object, but it is implemented in the old manner through a dialog window. In the next release this could and should change. As for the Clipboard, the best solution is to put the Find... option in the program's window context menu, and then display a dialog wherein the search specifications can be stated.



## Changing the Look: Fonts and Colors

The traditional approach for fonts is to present a special dialog window, as implemented in E.EXE. This is not a good solution and should be avoided. There is a system object known as the Font Palette, specialized in categorizing and displaying fonts that can assign them to windows by means of a drag & drop action. Inevitably, in all demonstrations of OS/2 this feature is displayed, although it is never actually used. It's time to change this. The Font Palette is a system-level tool, and, as such, is always available to and accessible from any application. Furthermore, the user will already know how to use it (it is an amusing operation for beginner OS/2 users to play with). The problem with the Font Palette, and also with the Color Palette, is that they are WPS object-generated starting from SOM classes. PM's API offers some tools for interacting with WPS objects directly within your own C code, with no great problems. This is the way to go. In practice, using a Font Palette and a Color Palette can completely replace the Options top-level menu in E.EXE. More precisely, all elements that take part in changing the look of an application (like displaying accessory windows, setting parameters like units of measurement in a DTP package, and so on), are best placed in one or more pages of a setting's notebook, just as in WPS.

Let's now examine the PMEDIT sample, which is a basic framework for creating an editor well-integrated in WPS.

### PMEDIT

The initial look of PMEDIT (Figure 13.23) is that of an ordinary window of the class WC\_MLE, overlapping the client of a generic frame. An initial difference is the absence of the client. It is the *mle* itself that plays the role of client for the main window.

By pressing the right mouse button in any portion of the *mle* you cause the window context menu to be displayed, and the window emphasized (Figure 13.24). This last aspect is important because it visually enforces the relationship between the menu and the object.

The absence of a client belonging to some class registered in the code, and the appearance of a window context menu associated with the *mle*, are two hints that demonstrate the need for subclassing both windows. The interception of messages addressed to the window procedure of the class WC\_MLE in some function given by the application also presents the possibility of setting up some code capable of dealing with drag & drop operations. The dragging of an object (in ASCII format) over the *mle* should be handled directly by the *mle* window as soon as the right mouse button is released (Figure 13.25).

PMEDIT also has a notebook for assigning some settings regarding the window. It is activated by selecting the Settings option from the secondary menu introduced by Open in the window context menu. The example shown in Listing 13.4 implements only a few pages (Figure 13.26): You can enrich the notebook with functionalities of your own.

The Attributes page shows the radio buttons that will set the color for the text, background, and font. In the page, there is no color map (like the valueset proposed in Listing 7.14), nor is there any list of available fonts. By pressing the radio button



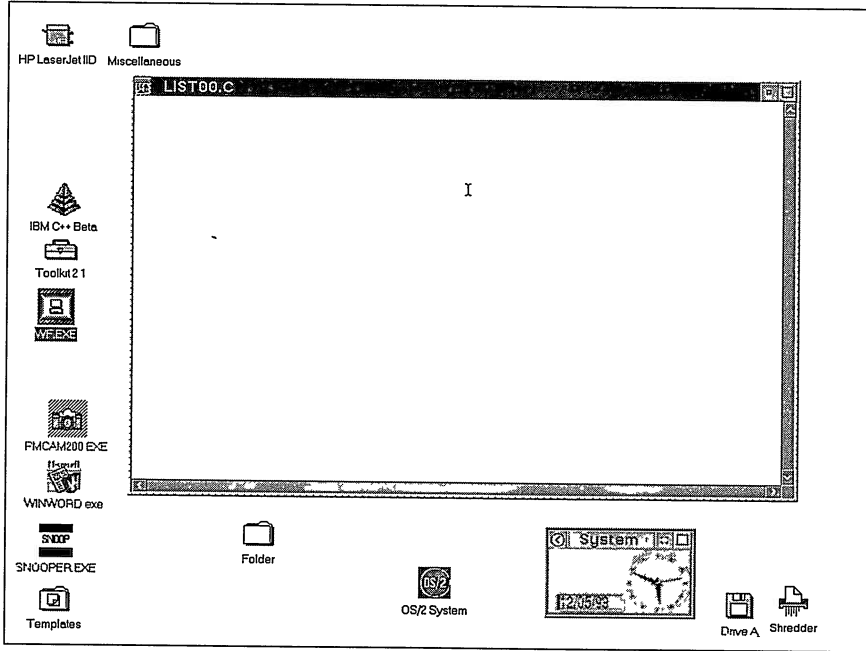


Figure 13.23 The PMEDIT editor is built around an *mle* window.

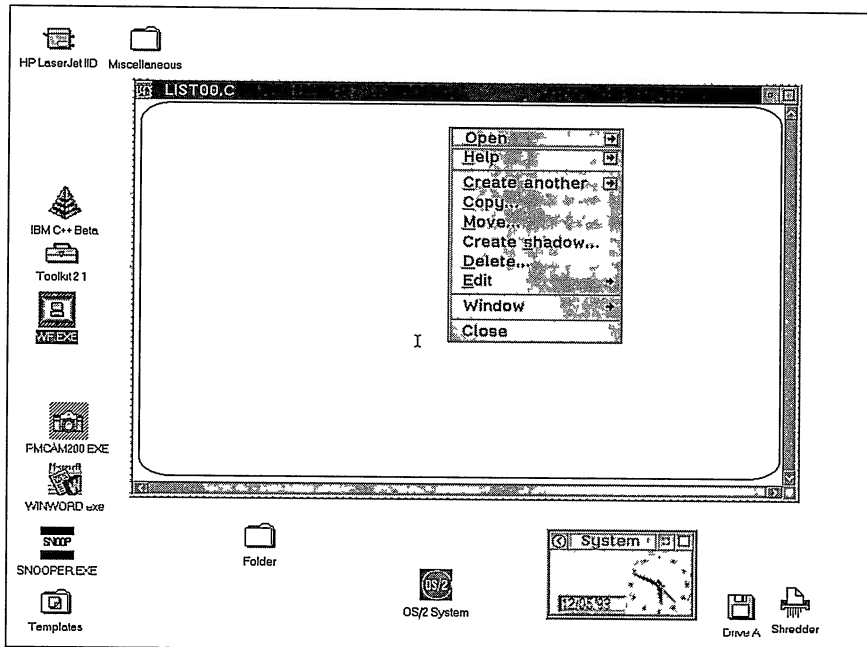


Figure 13.24 The editor is equipped with a *window context menu*.

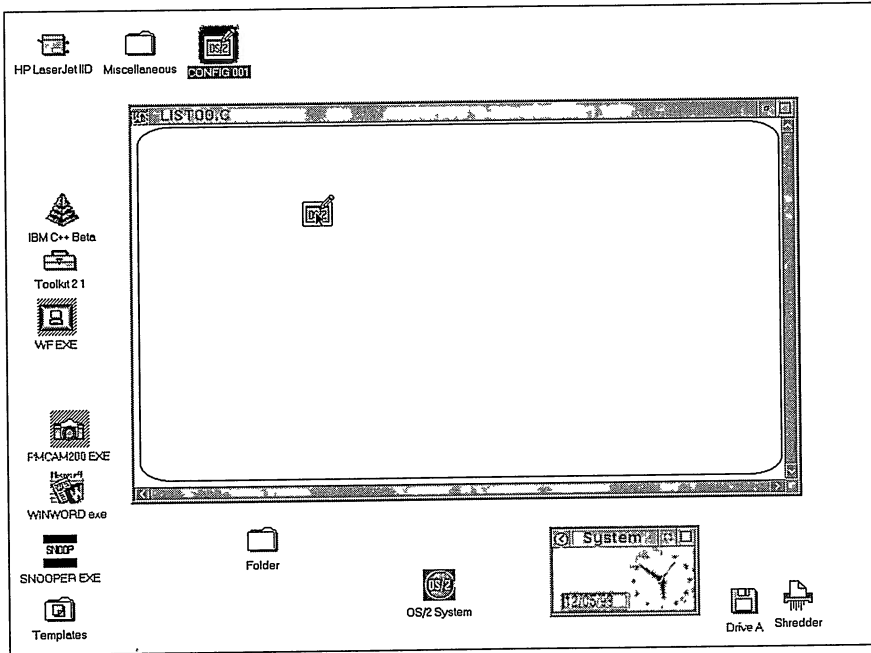


Figure 13.25 PMEDIT interacts with WPS by accepting objects in ASCII format.

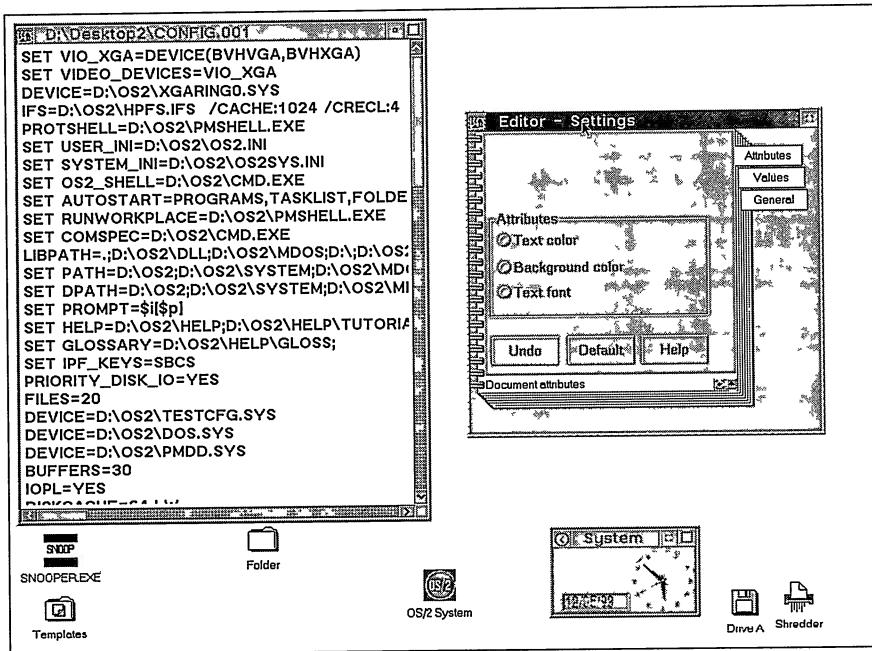


Figure 13.26 The Settings notebook of PMEDIT.

pertaining to the background color an instance of the Color Palette appears on the screen (Figure 13.27). Notice that the title, though, is different from that of the object present in the System Setup folder (Object Color in PMEDIT).

Select one color and drag it over PMEDIT. The editor adapts and uses it to paint its background. (However, there are some problems with the WC\_MLE class that requires a solution at the code level of its window procedure; only some of the primary colors will produce the desired effect on an *mle*. The custom control MIECC.DLL in Chapter 10 shows how to bypass the problem). The same logic extends to the setting of the color of text. After pressing the appropriate radio button, repeat the same operation in the Color Palette, and drag it to the editor; now it affects the text color (Figure 13.28). Naturally, even the change of font follows the same scheme (Figure 13.29).

PMEDIT supports various drag & drop operations, has a window context menu, and follows the SDI concept. These are all interesting features, which have partly been described in the previous examples. The real news is in the interaction with WPS objects (SOM).

## WPS Objects in PMEDIT

WPS objects used in PMEDIT are instances of equivalent objects present in the System Setup folder. Furthermore, the application will maintain a "private" use of its own

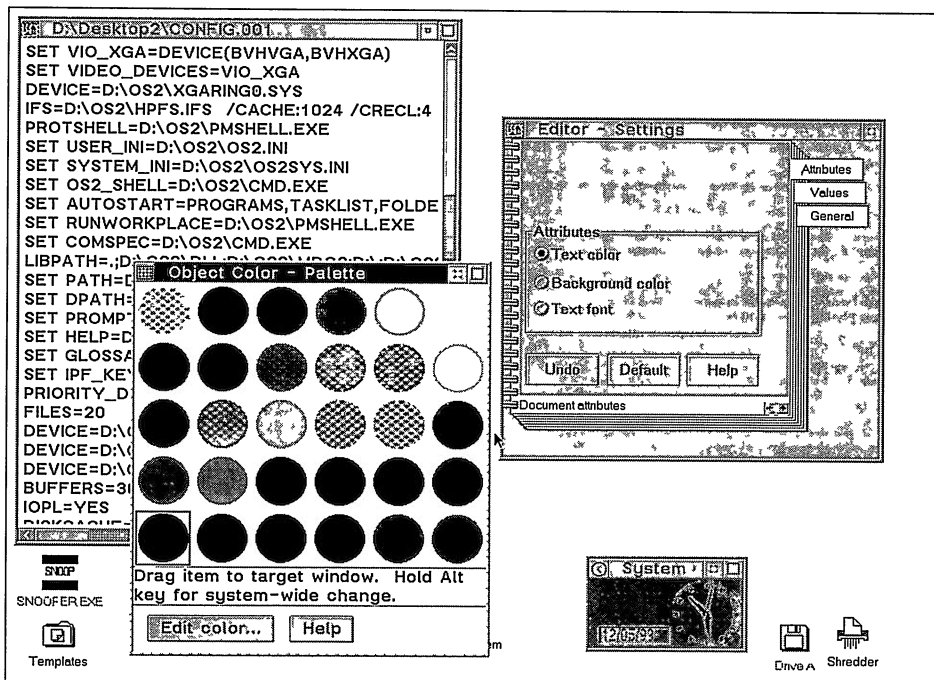


Figure 13.27 Setting of colors in the editor will respond to objects given by WPS.

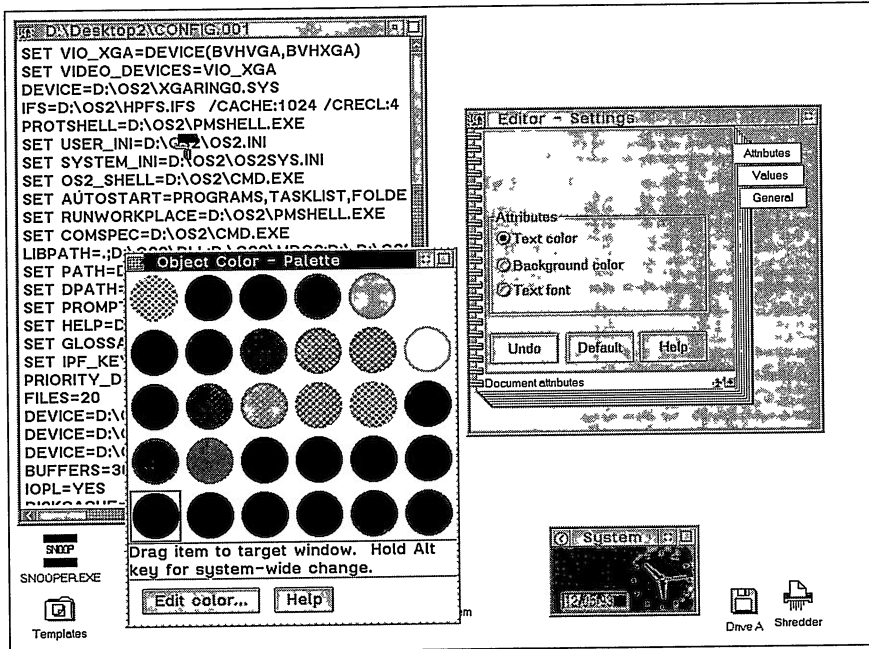


Figure 13.28 Assigning a color to the text present in the editor.

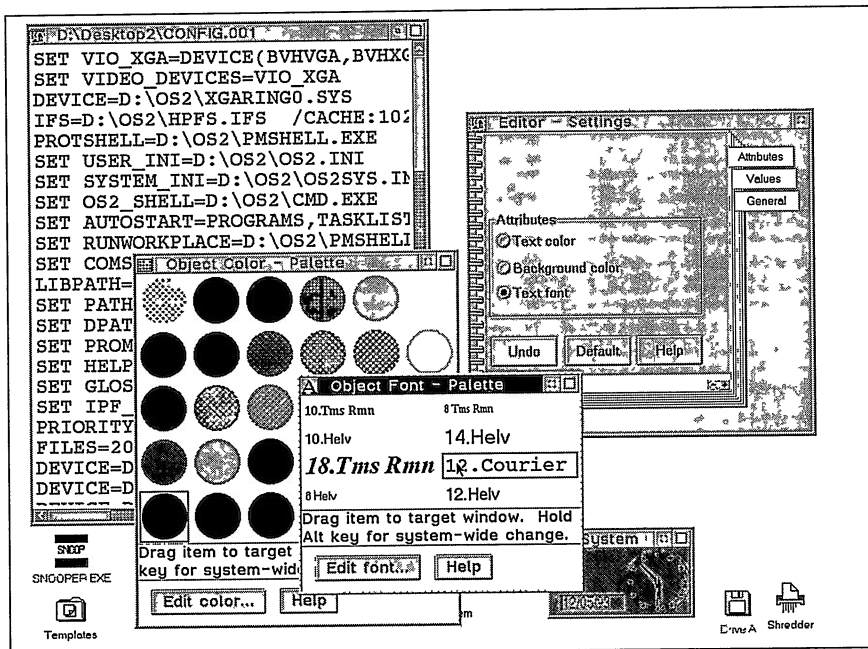


Figure 13.29 Assigning a new font to the document present in the editor.

instance of the Color Palette and of the Font Palette. Despite this, nothing prevents the user from coloring the background of the screen by taking the color from those present in the Object Color. However, the minimization or destruction of the Settings window in PMEDIT causes the simultaneous destruction of the WPS objects previously created by pressing the corresponding radio buttons. The integrated management of the Settings window and those of objects is controlled from within the application. Immediately after the creation of a system object, PMEDIT retrieves the window's handle and stores it in a data area of its own. The header file PMEDIT.H contains a new data structure, NBKDATA, defined like this:

```
typedef struct _NBKDATA
{
    HWND hwndClr ;
    HOBJECT hobjClr ;
    HWND hwndFnt ;
    HOBJECT hobjFnt ;
    BOOL fText ;
} NBKDATA ;

typedef NBKDATA * PNBKDATA ;
```

The first four members describe the pair of handles HWND/HOBJECT for the Color Palette and the Font Palette activated by the user. In order to save screen space and to simplify the user interface, the selection of a color for the text and for the background causes the appearance of only one single instance of the Color Palette. The boolean member fText takes the value TRUE when the color palette regards the text color. This does not directly affect the operations of creating a new object, but rather its subsequent use.

To retrieve the window handle of an object, the program employs a somewhat weak algorithm. The only working solution is to search for a window that has as its titlebar the string used when the *WinCreateObject()* was called. This is certainly not a fast and efficient approach. A better solution would be that of having a special API function for this purpose, but it is not available at this stage.

## *The Structural Elements of PMEDIT*

The application fits into a window that has an *mle* as its client. As usual, this window is subject to three operations: subclassing, assignment of the owner, and change of the background color.

Subclassing is imposed even on the titlebar menu icon, in order to support drag & drop operations. The new function is called *TitleMenuWndProc()*. Immediately after creating the main window and its ensuing subclassing, the program creates a second standard window having a *notebook* as its client. The notebook's handle is stored in the QWL\_USER portion of the application's reserved memory area (Table 13.5).

Both the *mle* and the *notebook* have as their owner their respective frame windows, even if they are the client (clients do not have owners). This variation on the traditional

**Table 13.5 The Window Procedures of the Classes and Dialogs Used in PMEDIT**

<i>Function</i>	<i>Description</i>
NewMleWndProc	Window procedure of the application's <i>mle</i> window
TitleMenuWndProc	Window procedure of the titlebar menu
NotebookWndProc	Window procedure of the notebook window
TitlebarWndProc	Titlebar window procedure to implement the title editing.
FrameWndProc	Frame window procedure to intercept drag & drop messages
ParamsWndProc	Dialog procedure of the dialog associated with the Values page of the Settings notebook
SettingsWndProc	Dialog procedure of the dialog associated with the Attributes page of the Settings notebook

scheme is accounted for by the possible need for intercepting the notification codes generated by the two windows belonging to predefined classes. Often, though, it is necessary to act directly at the subclassed window procedure level, because many classes do not have notification codes that are "smart" enough, especially concerning the typical WPS operations, like drag & drop.

## *Changing the Name of a Document*

PM EDIT offers yet another suggestion to OS/2 software application developers. In WPS, changing the name of an object is achieved by clicking the mouse button while the ALT key is pressed (or with the combination Shift+F9). In any case, this action is always performed directly on the object. By following this rule, we can effectively implement an alternative to the Save As... command typically found under the File menu. The titlebar is the item bound to contain the title of a window (application) and is also the principal tool for performing movements. The interaction between the user and a window of the class WC\_TITLEBAR is rather limited and well defined. By exploring the APIs of OS/2 you might come across the message WM\_TEXTEDIT, which is generated automatically by the system every time the user presses the left mouse button together with the ALT key. The WM\_TEXTEDIT message is sent to the window underlying the mouse's hot spot. This behavior suggests subclassing the titlebar in order to intercept WM\_TEXTEDIT. When the message WM\_PASSPROC is detected, the application will also create a window of the class WC\_ENTRYFIELD, with no size, and origin in the point (0, 0). The parent of this window will be the titlebar. Even the entryfield is subclassed so that the ENTER key can be detected.

		<i>Description</i>
WM_TEXTEDIT	0x0426	
mp1	USHORT usPointer	Source of the message: TRUE indicates the mouse, while FALSE indicates the keyboard
mp2	POINTS ptspointerpos	Pointer position expressed in window coordinates
Return Value	BOOL fResult	Indicates whether the message has been processed (TRUE) or not (FALSE)

The information contained in mp1 and mp2 of WM\_TEXTEDIT are not useful in this case. When the message is received, the application first retrieves the size of the titlebar, and then gets its title, with *WinQueryWindowRect()* and *WinQueryWindowText()*, respectively. The two pieces of information are used for sizing the entryfield so that it completely covers the titlebar; it is then displayed to the user, containing the application's title (Figure 13.30).

The loss of focus or pressing the ENTER key are the two conditions that terminate the window's titlebar editing, and makes the entryfield disappear from the screen.

This simple integration to the code of PM EDIT will make window management consistent with what is standard with WPS objects, and demonstrates once again how malleable and flexible the OS/2 user interface is.

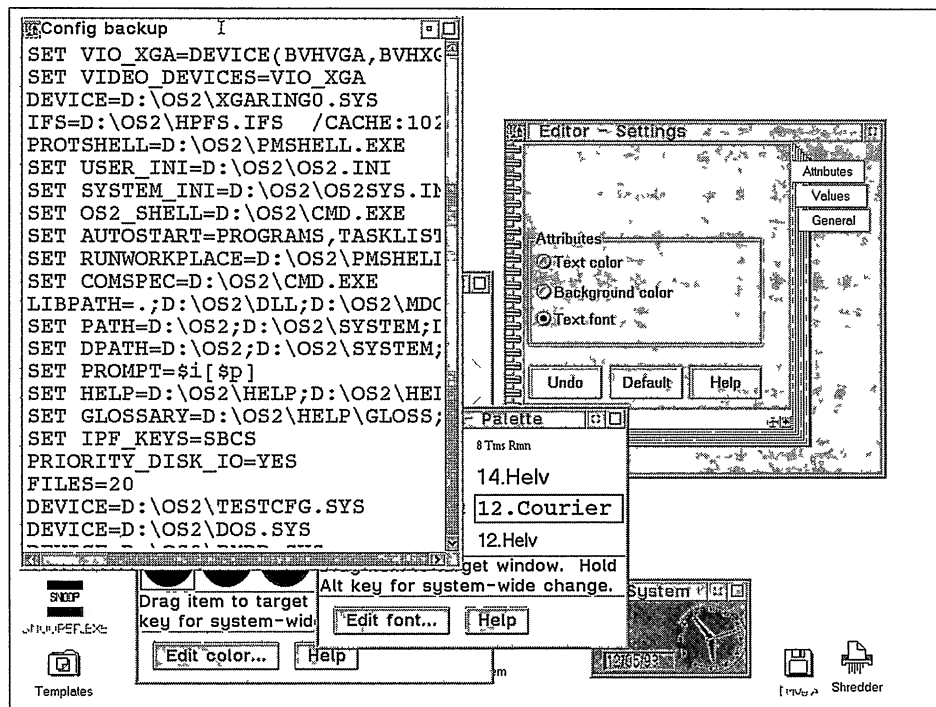


Figure 13.30 Editing the title of a window directly in the titlebar.



# Index of Listings

<i>Listing</i>	<i>Name</i>	<i>Description</i>
2.9	MACHINE	The source code file MACHINE.C.
3.3	ERASE	Erasing the client window with <i>GpiErase()</i> .
3.4	SHOW	This source code now allows you to color the client window in white.
3.5	CREATE	A WM_CREATE case is added in the window procedure of the application shown in Listing 3.4.
3.6	CLIENTCL	CLIENTCL controls the color of its client, changing it when the window is resized or when it gets the input focus.
3.7	PAINT	The source code of PAINT illustrates the output clipping mechanism of PM.
4.1	PARENT	A pair of windows both generated by calling <i>WinCreateStdWindow()</i> .
4.2	OWNER	Example of ownership between top-level level windows in PM.
4.3	MDI	Generation of two related PM windows by means of the <i>WinCreateWindow()</i> function.
4.4	STORE	Storage of window information in the OS2.INI system file.
4.5	TWOWND	A sample window procedure for displaying the rationale governing the positioning of a predefined class window with respect to the application's client window.
4.6	CLCOLOR	Code for checking the contents of the Window List's listbox.
4.7	ENUM	Enumeration of the top-level windows to detect the presence of any previous instance.
4.8	BROADCAST	Usage of <i>WinBroadcastMsg()</i> to prevent more than one instance from running at the same time.
4.9	SETFOCUS	The code used to intercept the WM_SETFOCUS message in a window procedure.

- |      |           |   |
|------|-----------|---|
| 5.1  | KBD       | The source code of a PM application that handles keyboard input and echoes it to the screen.                        |
| 5.2  | MOUSE     | The code that displays the mouse's coordinates inside the client area of a window.                                  |
| 5.3  | TIMER     | Sample source code of a program demonstrating the usage of timers in PM.  |
| 5.4  | WINTMP    | Code for creating a window starting from a window template in a resource file.                                      |
| 5.5  | ICON      | Assigning an icon to a window.  |
| 5.6  | SYSBMP    | Application that displays all of PM's predefined bitmaps.   |
| 5.7  | ICONDRAG  | Drawing and dragging an icon on the screen.   |
| 5.8  | BITMAP    | Source code of an application that allows you to move a bitmap by selecting it with the mouse.                      |
| 5.9  | ASSOCT    | The source code of the ASSOCT application, a practical example of the usage of the ASSOCTABLE resource.             |
| 6.3  | MENUAPP   | The source code of the MENUAPP application with a menu bar and an appropriately modified window procedure.          |
| 6.4  | MENUAPP   | Determining the input tool used to select a menuitem.   |
| 6.5  | MENUAPP   | Changing a menu item's attributes dynamically in the ATTRS application.   |
| 6.6  | TWOMENUS  | The application TWOMENUS is equipped with two menu templates; the two menus are loaded alternatively in the window. |
| 6.7  | TWOMENUS  | An elegant solution for handling two or more menus in the same window.  |
| 6.8  | BMPMENU   | Structure of an application that uses bitmaps in place of ordinary text strings as menu items.                      |
| 6.9  | ACCEL     | Dynamic loading of an accelerator table in ACCEL.   |
| 6.10 | RTMENU    | Building menus inside the client area of a window.  |
| 6.11 | CNTXT     | Source code of the CNTXT application with a customized titlebar menu and a window context menu.                     |
| 7.1  | SNOOPER   | SNOOPER, a PM application that can display the properties of windows of other programs.                             |
| 7.2  | BUTTONS   | The various kinds of button shapes present in the API of OS/2 PM.   |
| 7.3  | BUTTONS   | Changing the color of the client area of a window by using a pushbutton.  |
| 7.4  | SCROLLBAR | Usage of a horizontal scrollbar to modify the color of a window's client area.                                      |

7.5	LISTBOX	A sample listbox filled with text strings coming from a resource file: The selected item is reproduced in a window of class WC_STATIC.
7.6	FLOW	Capturing the message flow during window creation.
7.7	ODLIST	The insertion of a graphical objects in a listbox by delegating to the application all output operations.
7.8	ENTRY	Creating of an entryfield window that accepts a preset number of characters.
7.9	COMBO	A Combobox example.
7.10	NOTEBOOK	A simple notebook equipped with several primary and secondary page tabs.
7.11	FOLDER	FOLDER is an example of usage of a WC_CONTAINER class window in a PM application.
7.12	SLIDER	The SLIDER application demonstrates the usage of slider controls.
7.13	SPIN	The SPIN application shows how to insert numeric values or text strings in a spinbutton.
7.14	VALUESET	VALUESET shows all the functionality of a window of the class WC_VALUESET, with the exception of drag & drop operations.
8.1	PRESPARAM	Sample usage of the presentation parameters.
8.2	WHEREIS	The source code of the WHEREIS application for deleting a set of files.
8.3	OLDOPEN	The source code of a PM application presenting a modal dialog.
8.4	NEWOPEN	The code of the NEWOPEN example.
8.5	MODELESS	An example of the usage of a modeless dialog in PM.
9.1	MENU	The source code of Menu Maker.
9.2	SNOOPER	Source code of SNOOPER.
9.3	SNOOPER	High priority SNOOPER
9.4	WHEREIS	A multithreaded version of WHEREIS .
10.1	MSGQUEUE	Sample that displays the contents of an application's message queue.
10.2	SUBCLASS	An example of subclassing a control in PM.
10.3	SUPERCLS	Source code of an application performing superclassing.
10.4	BORDER	The source code of the BORDER application that produces the application shown in Figure 10.14.
10.5	MIE01	The source code of the MIE01.DLL dynamic linking library that provides its services to the BORDER application.

- |       |          |   |
|-------|----------|---|
| 10.6  | WINDOW   | The source code of the WINDOW.EXE containing the code registering a new class of windows.     |
| 10.7  | MIE02    | The source code of MIE02.DLL, a sample DLL used in the previous example.                      |
| 10.8  | SYSDLLS  | SYSDLLS.EXE lets the user selectively add and remove DLLs in the SYS_DLLS entry in OS2.INI.   |
| 10.9  | MIECC    | The source code of MIECC.DLL, a custom control automatically loaded at system bootstrap.      |
| 10.10 | USECTL   | USECTL creates one window of class TWENY registered in the MIECC.DLL custom control.          |
| 11.1  | CLIPPUT  | The application CLIPPUT illustrates the rules for transferring information to the Clipboard.  |
| 11.2  | CLIPSHOW | The application CLIPSHOW illustrates the rules for retrieving information from the Clipboard. |
| 11.3  | CLIENT   | A sample DDE client.  |
| 11.4  | SERVER   | A sample DDE server.  |
| 12.1  | DRAG     | An example of dragging.   |
| 12.2  | DROP     | DROP shows how to intercept and then accept an object via drag & drop.                        |
| 12.3  | PANEL    | The source code of the PANEL application.   |
| 12.4  | WPSFLDR  | The source code of WPSFLDR.   |
| 13.1  | SNOOPER  | The source code of the SNOOPWPS application.  |
| 13.2  | CLASSES  | The source code of the application CLASSES, which lists the classes registered in WPS.        |
| 13.3  | INSTALL  | The source code of the INSTALL application.   |
| 13.4  | PMEDIT   | The source code of the PMEDIT editor.   |

# Index

32-bit processors, 483–84

## A

accelerators, 241, 287–92  
    in WHEREIS application, 463  
ACCELTABLE resource, 287–92  
ACC files, 501–3  
action bar, 249  
anchor block, 37, 515  
API (Application Program Interface), 8–10  
ASSOCTABLE, 236–39  
asynchronous messages, 97, 98

## B

binary resources, 217–18  
bitmaps  
    as menu items, 285–86  
    moving, 233–36  
    predefined, 221–28  
        displaying, 226–28  
BKM\_messages, 383–85  
BKS\_styles, 382–83  
block scope, 38  
BTNCDATA structure, 323–26  
buttons, 320–29. *See also* pushbuttons;  
    WC\_BUTTON class windows  
    BTNCDATA structure and, 323–26  
    interacting with, 326–27

## C

CA\_attributes, 400  
cached micro PS, 83–85  
calling convention, 15  
CBM\_messages, 370  
CBS\_styles, 369–70  
CCS\_styles, 388–89  
child windows, 121–26  
    creating, 154–56  
C language, 12–19  
CLASSINFO structure, 312–14  
class registration of windows, 122–23  
CLIENT application, 598–606  
client window, 24  
ClientWindProc(), 66  
Clipboard, 571–82, 694  
    CLIPPUT and CLIPSHOW  
        applications, 578–82  
        examining the contents of, 578  
        inserting data into, 574–76  
        retrieving the contents of, 577–78  
        transferring an object with, 576  
    clipboard data formats, 572  
    Clipboard viewer, 578  
    CLIPPUT application, 578–80  
    CLIPSHOW application, 578–82  
CMA\_attributes, 407, 409–10  
CM\_ALLOCRECORD message, 393  
CM\_INSERTRECORD message, 398–99  
CM\_messages, 391–92

CM\_SETCNRINFO message, 401–2  
 CN\_notification codes, 407, 408  
 CODE directive, 490, 491  
 code segment, 487–88  
 colors, 695  
 comboboxes, 369–72  
 COMMANDMSG macro, 270–71, 276  
 COMMANDMSG structure, 276–77  
 compiling multithreaded applications, 514  
 containers. *See also* WC\_CONTAINER  
     class windows  
         creating, 392–93  
         display modes of, 389, 390, 401  
         objects of, 393–402  
         proliferation of objects and, 406–7  
 context switch, 37  
 CREATESTRUCT structure, 144–45  
 CS\_CLIPCHILDREN, 42  
 CS\_CLIPSIBLINGS, 42  
 CS\_FRAME, 42  
 CS\_HITTEST, 42  
 CS\_MOVENOTIFY, 42  
 CS\_PARENTCLIP, 42  
 CS\_PUBLIC, 42  
 CS\_SAVEBITS, 42  
 CS\_SIZEREDRAW, 42  
 CS\_SYNCPAINT, 42  
 CUA 89 (Common User Access 89), 21, 23  
 CUA 91 (Common User Access 91), 21,  
     23–24

## D

DATA directive, 489–91  
 data segment, 487–88  
 data types, 15–17  
 DDE conversations, 584–96  
     designing, 584–85  
     features of, 585–86  
     initiating, 586–90  
     invisible windows and, 596–97  
     permanent links and, 595  
     providing data to the client in, 593–95  
     requesting data in, 590–93  
     terminating, 595–96  
     DDE (Dynamic Data Exchange), 582–606  
         CLIENT application using, 598–606  
         uses of, 598  
     debugging, 19–20. *See also* IPMD debugger  
     DEF file (definition file), 30–31, 490–91  
         of a DLL, 554–58  
     defines, 15–16  
     deleting files, in WHEREIS application,  
         465–68  
     designing OS/2 applications, objectives  
         of, 659–60  
     desktop window, 120  
     destroying an object, 691  
     device context, 82, 83  
     device drivers, 81–82  
     DevOpenDC(), 86  
     dialog procedures, 429, 439, 440  
     dialog windows (dialogs), 124, 429–81  
         accessing controls in, 446–48  
         creating, 432–35  
         default message processing and, 457–58  
         features of, 430–31  
         as frame windows, 430, 432  
         modal, 430, 435  
         modeless, 430, 435, 445, 480–81  
         Open box, 469–75  
         ownership relationship and, 443–46  
         predefined, 476–79  
         presentation parameters of, 448–58  
             resource files and, 456  
             setting the presentation parameters,  
                 452–54  
             WinCreateWindow() and, 454–56  
         templates for, 436–38  
         terminating, 456–57  
         WHEREIS application, 458–68  
         WM\_INITDLG message and, 441–48  
     discarding, as memory management  
         technique, 486  
     DisKillThread(), 514  
     DLGEDIT.EXE, 436–38  
     DLLs (dynamic link libraries), 45, 491, 504,  
         540–59. *See also* import libraries  
         advantages of, 542–44  
         coding, 551–52

- compiling, 554
- creating, 552–54
- creating a new control and, 559–60
- DEF file of, 554–58
- definition of, 540–41
- executing programs that access, 548
- loading
  - explicitly, 549–51
  - implicitly, 548–49
- producing, 541–42, 544
- relocation record and, 545–48
- DM\_DRAGOVER message, 625, 628–37
  - return value of, 634–36
- DoCommands(), 272
- DosAllocMem(), 485
- DosCreateThread(), 512–14
- DosError(), 463
- DosFreeModule(), 549–50
- DosFreeResource(), 210
- DosGetInfoBlock(), 36–37, 517–18
- DosGetPrty(), 518
- DosGetResource(), 209, 210
- DosLoadModule(), 504, 549
- DosQueryCurrentDisk(), 462
- DosQueryModuleHandle(), 550
- DosQueryModuleName(), 550
- DosQueryProcAddr(), 550
- DosQueryResourceSize(), 209
- DosResumeThread(), 514
- DosSetMem(), 485
- DosSetPriority(), 516–17
- DosStartSession(), 158
- DosSuspendThread(), 514
- DosWaitThread(), 514
- DRAG application, 639–44
- drag & drop messages, 77
- drag & drop operations, 197–98, 607–58
  - acceptance feedback and, 633–34
  - API involved in, 607–24
  - changing the cursor's look and, 631–33
  - DRAG application, 639–44
  - DROP application, 644–48
  - executing the drag, 623–24
  - folders and, 654–58
  - frame window-client window
    - relationship and, 636–37
  - functions involved in, 607–9
  - with icons, 228–33
  - listboxes and, 650–54
  - message flow in, 610–11
  - messages involved in, 625–26
  - mouse messages regarding, 194
  - preparing objects for dropping, 625–39
  - preparing the image for, 622–23
  - receiving objects, 637–38
  - return value of DM\_DRAGOVER
    - message and, 634–36
  - selecting objects and, 613–22
  - titlebar dragging, 638–39
  - valuesets and, 645–50
- DRAGIMAGE structure, 622–23
- DrawCircle(), 92
- DrawXMasTree(), 93–94
- DrgAccessDragInfo(), 627–28, 637–38
- DrgAddStrHandle(), 617
- DrgAllocDraginfo(), 614–16, 620–21, 625
- DrgDrag(), 623–24, 625
- DrgFreeDraginfo(), 638
- DrgGetPS(), 633–34
- DrgQueryDragitemCount(), 628, 638
- DrgQueryDragitemPtr(), 628, 638
- DrgQueryStringName(), 629
- DrgSetDragImage(), 631–32
- DrgSetDragitem(), 616–17
- DrgSetDragPointer(), 632
- DrgVerifyNativeRMF(), 630–31
- DrgVerifyRMF(), 630–31
- DrgVerifyTrueType(), 629
- DrgVerifyType(), 629
- DrgVerifyTypeSet(), 629
- DROP application, 644–48
- drop-down menus, 241
  - customizing the look of, 253–55
  - defining, 497–99
  - MENUIITEM directive and, 252, 253
  - style rules for, 256–57

## E

editor for OS/2 2.1, 691–93  
 E.EXE, 691–93  
 EM\_messages, 366–67  
 ENTRYFDATA structure, 368  
 entryfields  
   multiple line (mle), 372–80  
 entryfield windows, 365–68  
 erasing  
   a window's background, 94–96  
 ES\_styles, 365–66  
 EXPENTRY functions, 17–18

## F

FCF\_flags, 50–55, 130  
   WC\_FRAME class and, 143–44  
 FCF\_SYSMENU flag, 243–44, 246, 249  
 FID\_flags, 123  
 FILEDLG structure, 476–78  
 file extensions, 11  
 focus handling, 175–79  
 folders  
   drag & drop operations and, 654–58  
 fonts, 695  
   listing, 478  
 frame control window, 130–32  
 FRAME directive, 212–13  
 frame window, 119–20  
 FrameWndProc(), 143–44  
 FS\_styles, 49–55  
 full-screen applications (FS), 3  
 function prototypes, 35, 36

## G

General Protection Fault, 488  
 generic PM applications, 21, 24–25  
   main() function for, 65–66  
   sample, 67–71  
 GpiAssociate(), 86  
 GpiBox(), 303  
 GpiCharStringAt(), 315  
 GpiCreatePS(), 85, 86  
 GpiErase(), 79–80, 96

GPI functions, 82  
 GpiLoadBitmap(), 223, 359  
 GpiQueryFontMetrics(), 358

## H

handles, 13–15  
 header files, 12–13, 34–35, 35–36  
 HHH files, 501–5  
   parsing, 504–5  
 hot spot, 188  
 Hungarian Notation, 17

## I

ICON directive, 217–18  
 ICONEDIT.EXE, 219  
 icons  
   loading, 218–21  
   moving, 228–33  
   predefined, 221–28  
     displaying, 226–28  
 import libraries, 545–46  
   producing, 558–59  
 initialization of an instance, 37–38  
 input focus, 183  
 INSTALL program, 684–90  
 instances of programs  
   executing several, 488–89  
   execution of one single instance, 159–68  
     accessing the Window List, 160–63  
     enumerating top-level windows,  
       163–65  
   initialization of, 37–38  
 integrating OS/2 applications in WPS,  
   660–61  
 IPMD debugger, 169  
   messages and, 104–7

## K

keyboard, 183–88  
   menu access with, 253, 256  
 keyboard control codes, 185–86  
 KWIKINF.EXE, 20



## L

listboxes. *See also* WC\_LISTBOX class  
 windows  
 creating, 350–53, 358–59  
 drag & drop operations and, 650–54  
 drawing items in, 361–65  
 handling information in, 360–61  
 macros used to interact with, 341  
 multiple selection, 343–44  
 owner-drawn, 354  
 reserved memory area of items in, 342–43  
 sending LM\_messages to, 341–42  
 simple, 346–49  
 top-index item of, 340  
 LM\_messages, 339–42  
 LOADONCALL directive, 487, 489  
 local heap, 31  
 LS\_OWNERDRAW style, 354

## M

magic cookies, 132  
 main() function, 36–43, 67  
   for a generic PM application, 65–66  
 main template, 283  
 make files, 25–31  
 maximized window, in Menu Maker, 493  
 MB\_flags, 171–74  
 MBID\_flags, 175  
 MDI (Multiple Document Interface), 281  
 memory  
   allocation of, 485–87  
   reserved  
 memory blocks, moving, 486  
 memory buffers of windows, 41, 43  
 menu bar, 24, 249  
   interactions between client windows  
   and, 296–97  
 MENUITEM directive, 251–53  
 menu items, 241  
   accelerators for, 287–92  
   bitmaps as, 285–86  
   built by the application, 286–87  
   categories of, 257  
   changing dynamically, 278–81  
   owner-draw-type, 287  
   recognizing the source of a selection of,  
   275–78  
 MENUITEM structure, 505  
 Menu Maker, 492–511  
   adding a drop-down in, 508–9  
   adding a top-level in, 507–8  
   attributes of of menu items in, 506  
   bitmap handle in, 506  
   control panel of, 495  
   drop-down menu in, 497–99  
   empty menu bar in, 493–95  
   functions of, 510–11  
   ID of menu items in, 506  
   interface of, 493  
   loading the menu template in, 502–4  
   maximized window in, 493  
   MENUITEM structure and, 505  
   position of menu items in, 505  
   saving the menu template in, 501–2, 507  
   separator bar in, 499–500, 509  
   styles in, 505–6  
   submenu handle in, 506  
   submenus in, 500–501, 509  
   top-level menus in, 496  
 menus, 241–305  
   drop-down, 241  
     customizing the look of, 253–55  
     defining, 497–99  
     MENUITEM directive and, 252, 253  
     style rules for, 256–57  
   keyboard access to, 253, 256  
   loading new, 281–85  
   macros and, 281  
   parenthood and ownership of, 268–69  
   run-time, 292–94  
   style rules for, 256–57  
   top-level, 241  
     defining, 496–97  
     SUBMENU directive and, 252  
   window context, 195, 241, 247, 297–302,  
     403–6  
     emphasizing the client window and,  
     303–5

- menus (*continued*)
  - WPS, 294–96
- menu templates, 249–53. *See also* Menu Maker
  - changing attributes dynamically, 278–81
  - complex, 261–63
  - defining, 258–61
  - loading, 265–68, 502–4
  - load options, 251
  - memory management options, 251, 252
  - saving, 501–2, 507
  - syntax rules for, 264–65
- message boxes, 169–75
- message loop, 60–65
- message queue, 37, 60
  - creating, 39
  - destroying, 64
- messages, 75–117. *See also* WM\_PAINT
  - message categories of, 77
  - DDE, 582–83
  - flow of, 97–109
    - debugging, 104–7
    - defining new messages, 108–9
    - functions that use messages, 108
    - messages of predefined classes, 108
    - non-queued messages, 101–3
    - posting messages, 97–101
    - predefined window classes and, 308–9
    - queued messages, 97–99
    - WC\_LISTBOX class windows, 349–50
    - windows and window procedures, 107
  - output handling and, 92–94
  - passing, 137–38
  - posting, 97–101
  - sending, 103, 111–14. *See also* WinSendMessage()
    - WinSetWindowPos(), 60
- MESSAGETABLE, 203, 209–11
- MIA\_CHECKED, 255
- MIA\_DISABLED, 255
- MIA\_FRAMED, 255
- MIA\_HILITED, 255
- MIA\_NODISMISS, 255
- micro PS, 85–86
- mini-pushbutton, 261, 296
- MIS\_BITMAP, 254
- MIS\_BREAK, 255
- MIS\_BREAKSEPARATOR, 255
- MIS\_BUTTONSEPARATOR, 255
- MIS\_HELP, 254
- MIS\_MULTIMENU, 254
- MIS\_OWNERDRAW, 254
- MIS\_SEPARATOR, 254
- MIS\_STATIC, 255
- MIS\_SUBMENU, 254
- MIS\_SYSCOMMAND, 254
- MIS\_TEXT, 254
- MLECTLDATA structure, 379–80
- MLM\_messages, 373–77
- MLN\_notification codes, 378
- MM\_messages, 273, 274, 279, 281
- MM\_QUERYITEMTEXT message, 273
- MM\_SETITEMATTR message, 279–81
- mnemonic code, 256
- MNU files, 501–2
  - parsing, 504–5
- module definition files, 30–31. *See also* DEF file
- module name, 490
- mouse, 188–98. *See also* drag & drop operations
  - clicking the, 195
  - selecting objects with, 195–97
- mouse pointers, predefined, 224–26
- moving icons, 228–33
- MQINFO structure, 529–30
- multitasking, 5–7
- multithreaded applications, 483–527. *See also* Menu Maker
  - compiling, 514–15
  - creating, 511–14
  - memory allocation and, 485–87
  - priority classes and, 515–16
  - segmented applications, 487–92
  - WHEREIS, 519–27

## N

NEWOPEN application, 478–79  
 NMAKE.EXE utility, 27  
 normal PS, 86  
 notebooks, 380–88, 381. *See also*  
     WC\_NOTEBOOK class windows  
     associating information with a page  
     of, 387  
     dialogs and, 435–36  
     filling, 383–85  
     first try at using, 387  
     inserting a page in, 385–87  
     page tabs of, 383, 385

## O

Open box, creating a, 469–75  
     centering the dialog, 471–72  
     filling in controls, 472  
     input sources, 472–75  
     new data type, 470–71  
     positioning the dialog, 470  
     selecting a file, 475  
 OS2386.LIB, 545  
 owner-draw-type menu items, 287  
 ownership relationship, between  
     windows, 126–30

## P

PAINT application, 115–17  
 painting, 78–83  
     execution of, 115–17  
     synchronous, 91–92  
 parent windows, 120–26  
 permanent links, 595  
 PM applications, 4–5  
     software tools needed for writing, 11–12  
 PMEDIT editor, 695–702  
 PM screen group, 5–8  
 PMSHELL.EXE, 515  
 PMWIN.DLL, 316–17  
 PMWIN.H, WM\_MENU class windows  
     and, 282  
 PMWP.H, API of, 677–84

POINTER directive, 217–18  
 posting messages, 97–101  
 predefined window classes, 39–40, 43–44,  
     307–427  
     as control elements, 308  
     creating new, 560–70  
     creating windows of, 318–20  
     message flow and, 308–9  
     undocumented, 318  
 WC\_BUTTON, 320–29  
 WC\_COMBOBOX, 369–72  
 WC\_CONTAINER, 388–410  
 WC\_ENTRYFIELD, 365–68  
 WC\_LISTBOX, 338–65  
 WC\_MLE, 372–80  
 WC\_NOTEBOOK, 380–88  
 WC\_SCROLLBAR, 333–37  
 WC\_SLIDER, 410–17  
 WC\_SPINBUTTON, 417–23  
 WC\_STATIC, 329–32  
 WC\_TITLEBAR, 332–33  
 WC\_VALUESET, 424–27  
     when to create windows of, 320  
     window words and, 317  
 preemptive multitasking, 511. *See also*  
     multitasking  
 PRELOAD directive, 487, 488  
 Presentation Manager. *See also index*  
     *entries starting with "PM"*  
 presentation parameters, of dialogs, 448–58  
     resource files and, 456  
     setting the presentation parameters,  
     452–54  
     WinCreateWindow() and, 454–56  
 presentation spaces, 82–86  
     cached micro, 83–85  
     device context distinguished from, 83  
     micro, 85–86  
     normal, 86  
     types of, 83–84  
 PrfQueryProgramHandle(), 158  
 printing, 694  
 priority classes, 515–19  
     selecting, 518–19  
 private data area, 31

processes, 5  
 process identification number (PID), 513  
 project files, 25, 30  
 protected mode, 488  
 public classes, registering, 168–69  
 PU\_flags, for window context menu,  
 300–301  
 pushbuttons, 327–29

## Q

QMSG structure, 61, 63  
 queued messages, 97–99  
 QW\_flags, 125

## R

RC files. *See* resource files  
 RECORDCORE structures, 393–98  
 RECORDINSERT structure, 397–98  
 registering window classes, 39–41, 45–46  
   public classes, 168–69  
 relocation record, 545–48  
 reserved memory area of windows, 132–36  
   extending, 135–36  
   indexes used to access information in,  
   132–33  
   querying, 136  
 resource files, 31–34, 200–215  
   binary resources in, 217–18  
   FRAME directive in, 212–13  
   icons in, 218–21  
   menu templates in, 258–61  
   MESSAGETABLE in, 209–11  
   presentation parameters and, 456  
   STRINGTABLE in, 203–9  
     defining computed IDs, 207–8  
     loading a string, 206–7  
     reasons for using, 208–9  
     syntax, 204  
   text resources in, 202–4  
   WINDOWTEMPLATE in, 211–17  
 resources, 200–218. *See also specific resources*  
   information characterizing, 201–2  
 run-time menus, 292–94

## S

saving  
   a document, 693–702  
   the position of a window, 153–54  
 SBM\_messages, 333–34  
 SBS\_styles, 333  
 scheduler, 5, 511, 512  
 screen groups, 5–7  
   switching among, 7  
 SCROLLBAR application, 335  
 searching and replacing text, 694  
 searching for a file, 459–61, 468  
 segmentation (segmented applications),  
 487–92  
   rules of, 491–92  
 SEGMENTS directive, 490, 491  
 selecting objects with the mouse, 195–97  
   mouse messages regarding, 194  
 semaphores, 168, 512  
 sending messages. *See also* WinSendMessage()  
 separator bar, 499–500, 509  
 sessions, 5. *See also* screen groups  
 sibling windows, 126  
 sizing icons, 24  
 sliders, 410–17. *See also* WC\_SLIDER class  
   windows  
     creating, 414  
     owner-drawn, 417  
     structural elements of, 413  
 SLM\_messages, 414–15  
 SLS\_styles, 411–12, 411–14  
 SM\_messages, 332  
 SNOOPER utility, 311–16, 661–68  
   priority classes and, 515–18  
 SNOOPWPS utility, 661–77  
   analysis of, 668–74  
   creating a panel for, 674–77  
 source code, 35–36  
 SPBM\_messages, 419  
 SPBN\_notification codes, 422  
 SPBS\_styles, 418  
 spinbuttons, 417–24  
   master-servant relationship of, 421–22  
 sample, 422–23

SPTR\_flags, 180–82  
 SS\_styles, 330–32  
 static linking, 544  
 StringParser(), 273  
 STRINGTABLE, 203–9, 347  
   defining computed IDs in, 207–8  
   loading a string from, 206–7  
   reasons for using, 208–9  
   syntax of, 204  
 subclassing, 136, 137, 532–37  
 SUBMENU directive, 251–53  
 submenus  
   adding, 509  
   defining, 500–501  
 superclassing, 537–38  
 SV\_flags, 147–51  
 swapping, 486  
   as a memory management technique,  
   486  
 swipe selections, 339  
 SWP\_flags, 152  
 synchronous, 247–49  
 system icons, adding, 179–82  
 system menu, 23–24  
   APIs of PM and, 248–49  
 System Object Module (SOM), 5, 678

## T

thread identification number (TID), 513  
 threads, 5, 511–14  
   creating, 512–14  
   maximum number of simultaneously  
   running, 6  
   PM applications and, 514–15  
 timers, 198–200  
 time-slices, 5, 518  
 titlebar icon, 22  
 titlebar menu, 243–48  
 titlebar(s), 22, 72, 332–33  
   dragging, 638–39  
 top-index item, 340  
 top-level menus, 241  
   defining, 496–97  
   SUBMENU directive and, 252

top-level windows, 121, 122  
   enumerating, 163–65

## U

update region, 87–88

## V

valuesets, 424–27  
   drag & drop operations and, 645–50  
 virtual keys, 289–91  
 VM\_messages, 426  
 VM\_SETITEMATTR message, 425  
 VN\_notification codes, 426  
 VS\_styles, 424

## W

WC\_BUTTON class windows, 43, 320–29  
   messages specific to, 326  
   notification codes specific to, 326–27  
   styles of, 321–22  
 WC\_COMBOBOX class windows, 43,  
   369–72  
 WC\_CONTAINER class windows, 43,  
   388–410. *See also* containers  
   attributes of, 400  
   CMA\_attributes of, 407, 409–10  
   messages of, 390–92  
   notification codes of, 406, 408  
   styles of, 388–89  
   window context menu and, 403–6  
 WC\_ENTRYFIELD class windows, 43,  
   365–68  
   ENTRYFDATA structure of, 368  
   messages of, 366–67  
   notification codes of, 367–68  
   styles of, 365–66  
 WC\_FRAME class windows, 43, 143–44  
 WC\_LISTBOX class windows, 44, 338–65.  
   *See also* listboxes  
   LM\_messages associated with, 339–42  
   message flow and, 349–50  
   notification codes used by, 345–46  
   styles of, 339

- WC\_LISTBOX class windows (*continued*)
  - WM\_DRAWITEM message and, 354–58
  - WM\_MEASUREITEM message and, 354
- WC\_MENU class windows, 44
  - styles available for creating windows of, 295–96
- WC\_MLE class windows, 44, 154–56, 372–80
  - data structures of, 377, 379
  - messages specific to, 373–77
  - notification codes of, 377, 378
- WC\_NOTEBOOK class windows, 44, 380–88. *See also* notebooks
  - notification codes of, 387
  - styles of, 382–83
- WC\_SCROLLBAR class windows, 44, 333–38
  - messages specific to, 333–34
  - styles of, 333, 334
- WC\_SLIDER class windows, 44, 410–17. *See also* sliders
  - messages of, 414–15
  - notification codes of, 416
  - styles of, 411–14
- WC\_SPINBUTTON class windows, 44, 417–24
  - messages of, 419
  - notification codes of, 422
  - styles of, 418
- WC\_STATIC class windows, 44, 329–32
  - messages specific to, 332
  - styles of, 330–32
- WC\_TITLEBAR class windows, 44, 332–33
- WC\_VALUESET class windows, 44, 424–27
- WHEREIS application, 458–68
  - accelerator table, 463
  - deleting files in, 465–68
  - error handling in, 463
  - executing a file in, 463–64
  - multithreaded, 519–27
  - scheme of, 461–62
  - searching for a file with, 459–61, 468
  - selection of a file in, 464–65
- WinAddSwitchEntry(), 157, 158, 163
- WinBeginEnumWindows(), 163–64
- WinBeginPaint(), 85–88, 92–94, 95
- WinBroadcastMsg(), 166–68
- WinCloseClipbrd(), 576
- WinCreateMsgQueue(), 39, 78, 514
- WinCreateStdWindow(), 46–56, 109–10, 119–22, 138, 140, 147
  - CREATESTRUCT structure and, 144–45
  - messages and, 105–7
  - parameters of, 47–48, 50
  - potential errors with, 56
- WinCreateWindow(), 46–47, 138–43, 147, 318–20
  - CREATESTRUCT structure and, 144–45
  - creating standard windows with, 140–43
  - experiments using, 144
  - FCF\_flags and, 143–44
  - parameters of, 138–40
  - presentation parameters and, 454–56
  - WC\_FRAME class and, 143–44
- WinDdeInitiate(), 586–90
- WinDdePostMsg(), 591–93
- WinDdeRespond(), 589–92
- WinDefDlgProc(), 457–58
- WinDefWindowProc(), 66–67, 75, 79, 146
- WinDestroyMsgQueue(), 64
- WinDestroyObject(), 691
- WinDestroyWindow(), 64, 145–46
- WinDismissDlg(), 456–57
- WinDispatchMsg(), 63, 66, 137–38
- WinDlgBox(), 443, 445, 458
- window classes
  - predefined. *See* predefined window classes
  - registering, 39–41, 45–46
  - public classes, 168–69
  - styles of, 41, 42
- window context menu, 195, 241, 247, 297–302, 403–6
  - emphasizing the client window and, 303–5
- WINDOW directive, 213–14
- windowed applications (WIN), 4
- Window List, 7
  - execution of one single instance of a program and, 160–63
  - informing, 156–59

- window procedures, 66–73, 529
  - accessing, 531–33
  - modifying, 269–75
- windows, 4–5
  - adding system icons to, 179–82
  - child, 121–26
    - creating, 154–56
  - class registration for, 122–23
  - closing, in multi-window applications, 146
  - creating, 46–56. *See also*
    - WinCreateStdWindow();
    - WinCreateWindow()
  - destroying, 145–46
  - dialog. *See* dialog windows
  - displaying, 56–60
  - focus of, 175–79
  - frame, 119–20
  - frame control, 130–31
  - getting information regarding, 310–16
  - message box, 169–75
  - ownership relationship between, 126–30
  - parent, 120–26
  - reserved memory area of, 132–36
    - extending, 135–36
    - indexes used to access information in, 132–33
    - querying, 136
  - resizing, 21
  - saving the position of, 153–54
  - sibling, 126
  - sizing and positioning, 147–53
  - top-level, 121, 122
    - enumerating, 163–65
- WINDOWTEMPLATE directive, in
  - resource files, 211–17
- window words, 137
  - predefined window classes and, 317
- WinDrawBitmap(), 223–24, 364
- WinDrawPointer(), 225–27
- WinDrawText(), 364
- WinEmptyClipbrd(), 574
- WinEndEnumWindows(), 163, 164
- WinEndPaint(), 85, 87, 92–94
- WinFillRect(), 80, 95, 112, 113
- WinFlashWindow(), 199, 200
- WinGetMessage(), 61–64, 66
- WinGetMinPosition(), 153
- WinGetNextWindow(), 163, 164
- WinGetPS(), 84–85
- WinGetSysBitmap(), 223
- WinInitialize(), 37
- WinInvalidateRect(), 89–92, 108
- WinIsWindowVisible(), 134
- WinLoadDlg(), 214–15
- WinLoadLibrary(), 206–7
- WinLoadMenu(), 250, 298, 502, 504
- WinLoadMessage(), 210–11
- WinLoadPointer(), 221
- WinLoadString(), 206, 208, 267, 315, 347
- WinMapWindowPoints(), 192, 193
- WinMessageBox(), 169–75
  - syntax of, 170
- WinMultWindowFromIDs(), 448
- WinOpenClipbrd(), 574
- WinPopupMenu(), 299–301
- WinPostMsg(), 99–101, 108–9
  - parameters of, 100
- WinQueryAnchorBlock(), 38, 161
- WinQueryCapture(), 191
- WinQueryClassInfo(), 312, 313, 352
- WinQueryClassName(), 164–65, 314
- WinQueryClipbrdData(), 577
- WinQueryClipbrdFmtInfo(), 577
- WinQueryClipbrdViewer(), 578
- WinQueryPointerPos(), 298–99, 362
- WinQueryPresParam(), 451–52
- WinQueryQueueInfo(), 529
- WinQueryQueueStatus(), 530–31
- WinQuerySwitchList(), 161, 162
- WinQuerySysPointer(), 180–82, 224–25
- WinQuerySysValue(), 147–53
- WinQueryTaskSizePos(), 154
- WinQueryWindow(), 123–26, 177
  - dialogs and, 443, 445
- WinQueryWindowPos(), 151
- WinQueryWindowProcess(), 157–58
- WinQueryWindowPtr(), 132, 531
- WinQueryWindowRect(), 80–81
- WinQueryWindowText(), 165

- WinQueryWindowULong(), 132, 134, 136
- WinQueryWindowUShort(), 132, 136
- WinRegisterClass(), 40–41, 45, 122–23, 135
- WinReleasePS(), 84–85
- WinRestoreWindowPos(), 153, 154
- WinSendDlgItemMsg(), 447
- WinSendMsg(), 101–3, 108–9, 137–38
  - dialogs and, 446, 447
- WinSetAccelTable(), 463
- WinSetCapture(), 189–91
- WinSetClipbrdData(), 574–76
- WinSetClipbrdOwner(), 576
- WinSetClipbrdViewer(), 578
- WinSetMultWindowPos(), 151
- WinSetOwner(), 128, 131
- WinSetParent(), 131, 268
- WinSetPresParam(), 448–49
- WinSetWindowBits(), 134, 135
- WinSetWindowPos(), 56–60, 71–72, 151, 154
  - HWND\_ definitions used with, 58
  - messages generated by, 60
  - SWP\_flags acceptable by, 58–59
- WinSetWindowPtr(), 132, 136, 535
- WinSetWindowText(), 72
- WinSetWindowULong(), 132, 135–36
- WinSetWindowUShort(), 131, 132, 135–36
- WinShowWindow(), 56–57, 107, 134
- WinStartTimer(), 199–200
- WinStopTimer(), 200
- WinStoreWindowPos(), 153
- WinSubclassWindow(), 534–35
- WinTerminate(), 64–65
- WinTrackInfo(), 234
- WinTrackRect(), 234–36
- WinUpdateWindow(), 90–92
- WinWindowFromID(), 123–26, 267, 446–48
- WinWindowFromPoint(), 192–93
- WM\_ACTIVATE messages, 111–13
- WM\_BEGINSELECT message, 196, 197
- WM\_BUTTON1MOTIONEND message, 196, 197
- WM\_BUTTON1MOTIONSTART message, 196–97
- WM\_BUTTONxCLICK messages, 195
- WM\_CHAR messages, 183–85
- WM\_COMMAND message, 269–73, 276–78
- WM\_CONTEXTMENU message, 195, 298–302
- WM\_CONTROL message, 345–46, 348–49
- WM\_CREATE messages, 109–10
- WM\_DRAWITEM message, 354–58
- WM\_ENDSELECTION message, 196, 197
- WM\_INITDLG message, 441–48
  - ownership relationship and, 443–46
- WM\_INITMENU message, 270, 273
- WM\_MEASUREITEM message, 354
- WM\_MOUSEMOVE messages, 188–90
- WM\_PAINT messages, 78, 79, 87–92, 115–17. *See also* messages
  - erasing a window's background and, 95–96
  - forcing, 89–91
  - output synchronization and, 91–92
  - update region and, 87–88
- WM\_SINGLESELECT message, 196
- WM\_SIZE messages, 113–14
- WM\_SYSCOMMAND message, 244–46
- WM\_TIMER messages, 198–200
- WM\_UPDATEFRAME message, 284, 285
- WPS objects, API for, 677–84
- WS\_GROUP style, 438
- WS\_styles, 48
- WS\_TABSTOP style, 438



# *How to Use the Disk*

---

## **Disk Contents**

The accompanying disk contains the following files:

BOOKINST.EXE	Installation program
MIE02.DLL	DLL required for installation
LH.EXE	Shareware compression utility
OS2WPS.LZH	Book examples
PRJS.LZH	Project files
README.TXT	Essentially, the same material as this page
LIST.TXT	Summary of program listings

---

## **Installation Procedure**

Start BOOKINST.EXE either from a full-screen OS/2 prompt or directly by double-clicking on the Drive A Workplace Shell object. BOOKINST creates a window on the screen with the book cover as its background. We strongly recommend that you run OS/2 in a high-resolution mode (SVGA 600 × 800 or better); if you don't, you'll be able to tell immediately at this point, because the bitmap of the cover will spill off the top of the screen and the colors will be, well, ugly.

Simply click the right mouse button on any location in the client window (book cover) to let the window context menu pop up. Choose install to start the installation procedure. A listbox appears at the bottom of the window showing all the available drives in the system. Select the target drive to start the installation procedure. When it terminates, you will be notified by a beep.

The files are divided in 13 directories, one for each chapter. Each chapter directory has a variable number of subdirectories, one for each code sample. BOOKINST creates the \PM32 directory on the target drive with a tree structure that resembles this:

```

z:\PM32
 |
 CHAP02
 |
  LIST01
    xxxxx0.yyy
    xxxxx1.yyy
    xxxxx2.yyy

```

where *z* is the target drive and *xxxxx0.yyy* is a generic file. BOOKINST also creates a shadow copy of the \PM32 directory on the desktop.

For each example, you also will find copies of the .MAK and the .DEP files. I strongly recommend you re-create both files if you experience any problem during compilation time or are using a different version of the C Set++ compiler (or another compiler).

You can always interrupt the installation procedure by selecting the Close menu item in the window context menu.

---

## Compilers

All the examples have been written and tested using the IBM C Set++ 2.0 compiler on an IBM PS/2 model 90 with 24MB RAM, 2 SCSI 400MB drives, and an XGA card. The installation procedure looks for the \IBMWF directory in which to appropriately store the .PRJ files for each example provided in the book. Later versions of this compiler might require a different location for the .PRJ files. For your convenience, all the .PRJ files are located in each subdirectory, too.

The examples provided don't have any specific dependencies on the C Set++ compiler. Recompile with other OS/2 C/C++ compilers requires the creation of only appropriate project files.



INTERMEDIATE/ADVANCED

IBM

OPERATING SYSTEM

## Design Applications for OS/2 2.1, IBM's Multithreaded Operating System for the PC

OS/2 2.1 is the latest release of IBM's multitasking environment for the PC platform. Using its Workplace shell, you can develop applications that stand out, not only for their user interface, but for their inner workings as well. *OS/2 2.1 Workplace Shell Programming* guides you through the process of building those applications, with examples throughout the text and source code supplied on the enclosed 3.5-inch disk. In addition, *OS/2 2.1 Workplace Shell Programming* covers:

- The OS/2 2.1 system architecture
- Memory management
- The graphics display interface
- OS/2 2.1 programming tools

Using OS/2 2.1 will enable you to enjoy the advantages of InterProcess Communications, a flat memory model, semaphores, and protected memory. The programming languages that can be used with OS/2 2.1 are those most commonly found on other hardware and software platforms, although OS/2 has a strong bias toward C and C++ because of their flexibility and power.

You'll find that the following features, among others, distinguish OS/2 2.1 from other operating systems:

- It has the ability to exploit all the power of a fully preemptive multitasking system, which means better performance and a higher level of protection for each task in execution.
- The Workplace Shell is the system interface as well as the first application that fully benefits from the system's API, the Presentation Manager, and the objects of the System Object Model, a language-neutral environment for defining, managing, and interacting with class libraries.
- OS/2 2.1 is simple and intuitive to use, thanks to the Workplace Shell, which is a new breed of object-oriented user interface.
- All IBM development tools are supported by rich on-line documentation, and are full of cross-references that make it a breeze to use.

**Stefano Maruzzi is a leading authority on OS/2 2.1, and is the author of several best-selling computer books in Italian.**



**RANDOM HOUSE**  
ELECTRONIC PUBLISHING

0-679-79162-0  
U.S. \$44.00  
Can. \$58.00

ISBN 0-679-79162-0



5 4 4 0 0



9 780679 791621